Thesis for The Degree of Doctor of Philosophy

Designing Trustworthy Autonomous Systems

Piergiuseppe Mallozzi

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2021

Designing Trustworthy Autonomous Systems

Piergiuseppe Mallozzi

ISBN 978-91-7905-506-6
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 4973.
ISSN 0346-718X

Technical Report No 198D
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

"Without a specification, a system cannot be wrong,
it can only be surprising!"
- Gary McGraw

Abstract

The design of autonomous systems is challenging and ensuring their trustworthiness can have different meanings, such as i) ensuring consistency and completeness of the requirements by a correct elicitation and formalization process; ii) ensuring that requirements are correctly mapped to system implementations so that any system behaviors never violate its requirements; iii) maximizing the reuse of available components and subsystems in order to cope with the design complexity; and iv) ensuring correct coordination of the system with its environment.

Several techniques have been proposed over the years to cope with specific problems. However, a holistic design framework that, leveraging on existing tools and methodologies, practically helps the analysis and design of autonomous systems is still missing.

This thesis explores the problem of building trustworthy autonomous systems from different angles. We have analyzed how current approaches of formal verification can provide assurances: 1) to the requirement corpora itself by formalizing requirements with assume/guarantee contracts to detect incompleteness and conflicts; 2) to the reward function used to then train the system so that the requirements do not get misinterpreted; 3) to the execution of the system by run-time monitoring and enforcing certain invariants; 4) to the coordination of the system with other external entities in a system of system scenario and 5) to system behaviors by automatically synthesize a policy which is correct.

Keywords

Autonomous Systems, System Trustworthiness, Formal Verification, Reinforcement Learning, Runtime verification, Monitoring and enforcement, Assume-Guarantee Contracts, Reactive Synthesis.

Acknowledgment

First of all, I would like to thank my supervisor Patrizio Pelliccione for believing in me and allowing me to embark on an incredible journey. Thank you also for giving me the freedom to explore different topics and find my way.

I would like to express my sincere gratitude to my co-supervisor Gerardo Schneider for welcoming me in the formal methods division and never giving up answering all my critical questions that have often lead to engaging in interesting discussions.

To my examiner Ivica Crnkovic for always being there for me where I needed support. Thank you so much also for giving me the opportunity and the responsibility to design and teach the machine learning course, it has been a truly rewarding experience.

I am deeply grateful to Alberto Sangiovanni-Vincentelli for inviting me to UC Berkeley and showing new perspectives on how research can be conducted. Thank you Antonio and Inigo for the helpful discussions during my stay there. I am looking forward to coming back to such a rich environment full of inspiration, ideas, and incredibly talented people.

I would also like to thank all my collaborators. In particular, I am grateful to Pierluigi Nuzzo for the constant support, for answering my doubts, and for helping me steer my research in new and exciting directions. Thank you also to Nir Piterman for your insightful comments and suggestions. I look forward to continuing our collaboration in the future.

This work would not have been possible without the support of the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation. I can not believe how lucky I am for being part of such an ambitious program. I will always be grateful for all the opportunities that WASP has offered me.

I want to thank all my colleagues at the Computer Science and Engineering department. In particular, I would like to thank my office mates Katja, Sergio, Rebekka, and Claudia for always making the office a fun place to work and for the deep discussions during lunch.

Finally, I would like to express my deepest gratitude to my family and all my friends. To my music friends in Gothenburg, thank you Enzo, Eric, and Max for jamming together and playing Take Five in all kinds of time signatures possible. Thank you also to Marco, Evgeni, Carlo, and Grischa for the gigs around the city and the good old times in the music room. Special thanks to Ron and Margaret for listening to our music. To all other Gothenburg people, in particular, Payam for the long discussions on life and politics; Romaric for

climbing together, and to always entertain everybody during our poker nights; Lydia and Tuğçe for inviting me to your nice parties and cooking amazing food; Tim and Francis for getting drunk together; Katja for the good old jazz and cigars nights; Giacomo, Luca, and Riccardo for making me feel like I was still in Italy. To Marta for sharing these last years of pandemic and fake lockdowns at home eating and cooking delicious food and never being bored. Thank you Giammi for often buying my groceries and for being who you are. To my '*japanese*' friends Veronica, Erik, Javi, Julen, Chris, and Anna and to Japan Society for the Promotion of Science (JSPS) for making our experience possible. Thank you, Ezequiel and Kenji for hosting me at Waseda University. To my '*french*' friends, particularly thank you Mancho for being a good friend and often beating me at chess. To my '*UK*' friends Lili and August for all our adventures around London back in the days. To my '*american*' friends, especially Sarah, Tommaso, and Andrea for making my stay there truly memorable and joining me on trips around the world. I can not wait to see you soon and start new adventures together! Last but not least, to my *Pisa* friends Giacomo, Ianne, Joy, Angelo, Pierf, Carols, Polli, Colzi and Michela for keeping in touch after all these years, for participating in the sometimes heated WhatsApp discussions, and for always welcoming back to Pisa. And to my *hometown* friends Federico, Stefano, Stefy, Micol, and Chiara for always trying to be there for our reunions. Special thanks also to Argo and Nina for always making me happy. I love you all, see you soon!

List of Publications

Appended publications

This thesis is based on the following publications:

A P. Mallozzi, M. Sciancalepore, and P. Pelliccione "Formal verification of the on-the-fly vehicle platooning protocol"
in *Software Engineering for Resilient Systems (SERENE), Gothenburg (Sweden)*, 2016, Springer

B P. Mallozzi, R. Pardo, V. Duplessis, P. Pelliccione, and G. Schneider "MoVEMo - A structured approach for engineering reward functions"
in *International Conference on Robotic Computing (IRC)*, Laguna Hills (CA), 2018, IEEE

C P. Mallozzi, E. Castellano, P. Pelliccione, G. Schneider, and K. Tei "A runtime monitoring framework to enforce invariants on reinforcement learning agents exploring complex environments"
International Workshop on Robotics Software Engineering (ROSE), Montreal (Canada), 2019, IEEE/ACM

D P. Mallozzi, P. Nuzzo, P. Pelliccione and G. Schneider "CROME: Contract-Based Robotic Mission Specification"
International Conference on Formal Methods and Models for System Design (MEMOCODE), Jammu (India), 2020, IEEE/ACM

E P. Mallozzi, P. Nuzzo, P. Pelliccione "Incremental Refinement of Goal Models with Contracts"
International Conference on Fundamentals of Software Engineering (FSEN), Tehran (Iran), 2021, IPM

F P. Mallozzi, G. Schneider, N. Piterman, P. Nuzzo, and P. Pelliccione "A Framework for Specifying and Realizing Correct–by–Construction Contextual Robotic Missions using Contracts"
in submission

Other publications

The following publications were published during my PhD studies. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

G P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha "Autonomous vehicles: state of art, future trends, and challenges" book chapter in *Automotive Software Engineering: State of the Art and Future Trends*, 2017, Springer

H P. Mallozzi, P. Pelliccione, and C. Menghi "Keeping intelligence under control" in *International Workshop on Software Engineering for Cognitive Services (SE4COG), Gothenburg (Sweden)*, 2018, ACM

I P. Pelliccione, E. Knauss, R. Heldal, S. M. gren, P. Mallozzi, A. Alminger, and D. Borgentun "Automotive architecture framework: The experience of Volvo cars" in *Journal of Systems Architecture (JSA)*, 2017, Elsevier

L P. Pelliccione, E. Knauss, R. Heldal, M. Ågren, P. Mallozzi, A. Alminger, and D. Borgentun "A proposal for an automotive architecture framework for Volvo Cars" in *Automotive Systems/Software Architectures (WASA)*, Venice (Italy), 2016, IEEE

M P. Mallozzi "Combining machine-learning with invariants assurance techniques for autonomous systems" in *International Conference on Software Engineering Companion (Doctoral Symposium)*, 2017, IEEE

Research Contribution

I took the lead in all the included papers.

The implementation, evaluation of the protocol and the formalization of the properties described in Paper A was done entirely by me. The remaining authors supported me with the formal verification of the properties in randomly generated scenarios and by reviewing the paper.

In Paper B I have designed the method, set up the experiment and collected and analyzed the data. The implementation was partially realized by me and partially by the third author. The remaining authors contributed to reviewing the paper, writing paragraphs related to the run-time monitoring and external tools support.

Most of the contribution of the Paper C is the result of my work. I have designed the method and implemented the approach. The remaining authors have helped me launching the experiments and collecting data.

I have designed the methodologies, implemented the tools and collected the results of Paper D, Paper E and Paper F. The remaining authors have helped me addressing theoretical doubts, and polishing the paper.

Contents

Chapter 1

Introduction

Autonomous vehicles, unmanned aerial vehicles (UAV), autonomous trading systems, self-managing telecom networks, smart factories can be all considered Autonomous Systems. They are becoming ubiquitous in our society and in the near future, we will assist in systems exhibiting higher and higher levels of autonomy that will put new demands on the engineering of such systems.

Autonomous Systems are characterized by the *interaction* between the system and its environment and how they respond to changes in their environment. The interaction can be physical, like in Cyber-Physical Systems (CPS), i.e. systems that rely on the seamless integration of their digital components with the physical world (e.g. autonomous vehicles). The interaction with the environment can also be completely digital (e.g. autonomous trading systems). The response of the autonomous system to changes in their environment without human interventions determines the degree of *autonomy* that the system has.

Some communities refer to autonomous systems as *self-adaptive systems*, i.e. systems that can adapt their behavior at runtime without human intervention [1–3] in response to changes in the environment or their internal state. They implement some sort of feedback loop to perform their adaptations [4]. A self-adaptive system gathers observations from the environment, processes them considering also its internal state, and adapts itself to achieve its goals.

Self-adaptive systems implement some sort of feedback loop that drives their adaptations [4]. This basic mechanism for adaptation has been applied for years in control engineering; it consists of four main activities: collect, analyze, decide, and act.A feedback loop particularly important for decision-making is the OODA loop by Boyd [5] where we do not only predict what our system has to do but also what the external systems are going to be doing.Another well-known reference model for describing the adaptation processes is the MAPE-K loop (consisting of the parts Model, Analyse, Plan, Execute, and the Knowledge Base) [6]. Furthermore, often systems collect data from the environment, learn from them, and, consequently, continuously improve. An example of such a system is the Never-Ending Language Learning [7].

System design is the process that, starting from some problem and going to several phases, ultimately provides a system that is the solution to the identified problem. We consider the three major phases of system design to be **modeling**, **design**, and **analysis** [8]. Modeling is the process of imitation of

the system in order to accurately describes the properties of our interest. In the design phase, we build the system through a structured process. Finally, the analysis phase aims at ensuring that the system behaves as it should. System design consists of iteratively moving from one phase to another in order to ultimately build the system.

A lot of problems in the design of autonomous systems concern how to deal with *uncertainties*. Sources of uncertainty could be *external* to the system, such as the environment in which the software is deployed, the availability of the resources that the system can access at a given time, or the difficulty of predicting the other systems behavior. Other sources of uncertainty can be *internal* to the system, such as the inability of the system to predict the impact of its actions on the environment. Uncertainty requires the system designer to carefully analyze the requirements, the environment assumptions, and the system specification. Often, uncertainties require the system to change and *adapt* at run-time or *evolve* at design-time.

Classical software development techniques require to fully describe the system behavior in each possible environmental condition. The developer models different configurations at design-time that are then activated at run-time according to the events coming from the environment [9]. This is becoming unpractical — if not even impossible — in AS, where there is a high, or even infinite, number of environmental conditions to be considered for adaptation. For this reason, modern development techniques for AS must rely on techniques that allow creating systems that autonomously *learn* how to behave in different environmental conditions.

Reinforcement Learning [10] has been successfully used as a technique to act in an unknown environment to achieve a goal. It enables the software agent to autonomously learn to perform actions by *trial and error*. As with self-adaptive systems, it is based on a feedback loop where the software agent performs actions on the environment in response to the observations, receiving a numerical value (*reward*) for each action. The goal of the RL agent is to maximize the cumulative received reward. After training, the RL agent can effectively handle changes i.e., when a change occurs, the system autonomously learns new policies for action execution.

The use of reinforcement learning as the main driver of the systems adaptation introduces flexibility in terms of *explore* and *acting* in an unknown environment but it also poses new challenges. The choice of which adaptation to perform in the environment is no longer in the developers hands but is rather automatic. When dealing with safety-critical systems one major challenge is the assurance *safety*.

Trustworthiness aims at establishing some degree of *trust* that the system is performing what it is supposed to do. Especially when dealing with safety-critical applications, it is essential to guarantee that AS act *safely* in the environment where they are deployed. In other words, the software must work reliably and must be safe for humans as life may depend on it. In this thesis, we refer to building *trustworthy autonomous systems* meaning to provide evidence that important aspects of the AS are correct. In presence of self-adaptation systems, the fulfillment of the requirements cannot be guaranteed completely at design-time but at least part of the assurance needs to be performed at runtime [11, 12].

In this thesis, we aim to provide *assurance* to the software system by combining design-time modeling and verification techniques. We leverage on the theory of *contracts of system design* for expressing a set of properties that the system components have to satisfy. Then, we use techniques such as *model checking*, *reactive synthesis*, and *runtime verification* to *i)* check that a model of the system satisfies its properties, *ii)* automatically synthesizes a model of the system, and *iii)* providing runtime assurances while the system is running, respectively.

Contract-based design [13, 14] has emerged over the years as a design paradigm capable of providing formal support for building complex systems in a modular way, rooted in sound representations of the system semantics and decomposition architecture. A *contract* specifies the behavior of a component by distinguishing the responsibilities of the component (*guarantees*) from those of its environment (*assumptions*). Contract operations and relations provide formal support for notions such as stepwise *refinement* of high-level contracts into lower-level contracts,*compositional reasoning* about contract aggregations, and *reuse* of pre-designed components satisfying a contract.

Model Checking [15, 16] is used to verify that a given model of a system respects some requirement specification which could be expressed, for example, with contracts. It works by exhaustively and automatically checking all the states of the system. The state space is a directed graph whose nodes encode the states of the whole system and whose edges represent the state change. It ultimately represents all the behaviors of the modeled system by branching all possible ways the components can interact with each other. Because of the exploration of all states in a brute-force manner, model checking suffers from the *scalability* problem.

Reactive Synthesis [17, 18] is used to automatically generate a model of the system acting in an environment when the specification requirements are expressed using some discrete-time temporal logic (e.g. LTL). It can be viewed as a two-player game where on one hand we have the environment that attempts to falsify the specification and, on the other hand, we have the system that tries to satisfy it. The system implementation is a controller, given in the form of a finite-state machine, which reacts to any possible input assignment (controller by the environment) with some actions (controlled by the system) that do not violate its specification requirements as long the environment does satisfy its constraints. If such controller exists we say that the specification is *realizable*. Synthesis from an LTL specification is even more complex than verifying that a specification holds on a given model as with model checking. For a general LTL formula, the synthesis problem has a doubly exponential complexity with respect to the size of the formula, while for a subset of LTL, namely generalized reactivity (GR(1)) [19] the complexity is exponential.

Runtime Verification (RV) [20, 21] is a lightweight verification technique based on monitoring software executions. It has its origins in *model checking* but it mitigates the state explosion problem by having a more scalable approach to software verification. It detects violations of properties, occurring while the monitored program is running, eventually providing the possibility of reacting to the incorrect behavior of the program whenever an error is detected. One way to verify properties at runtime is through the use of *monitors*. A monitor is a piece of software that runs in parallel to the program under scrutiny, controlling

that the execution of the latter does not violate any of the properties. Besides, monitors usually create a log file where they add entries reflecting the verdict obtained when a property is verified. In general, monitors are automatically generated from the annotated/specified properties [22].

On one hand, static verification and synthesis approaches such as model checking and reactive synthesis can respectively verify that system is compliant with some important properties or synthesize it, but these techniques can be sometimes very difficult to use in practice due to their computational complexity. On the other hand, runtime verification can prove the satisfaction of the properties on large systems but only on the fraction that is executing during the verification.

Our research goes in the direction of building **trustworthy autonomous systems** with particular emphasis on the theory of contracts as a methodology to formalize the system requirements, reinforcement learning to drive the system adaptations under uncertainty and model checking, reactive synthesis, and runtime-verification as formal machinery to provide certificates of trustworthiness. In Section 1.1, we present the general problem framed with a *trustworthiness argument*, in Section 1.2 we present the challenges and related work associated in the satisfaction of the trustworthiness argument, in Section 1.3 we present the background on the formal machinery that we use to provide trustworthiness and drive the system adaptations, in Section 1.4 we present the goals methodology of this thesis and addressed research questions, and in Section 1.5 we present our contribution and a summary of the studies. Finally in Section 1.6 we present our conclusions and future work.

1.1 The world and the machine

A system might be seen as a *machine* that is expected to solve some problem in the real *world*. The interaction between the system and the world happens via an interface containing the *shared phenomena* between the machine and world. Through this interface the system *observes* some phenomena and *controls* others in order to solve some problem [23].

The **requirements** formulate what the system has to achieve in the world in order to solve some problem. Understanding what is the problem that needs to be solved can be extremely difficult. Requirements are concerns with what the system should do, possibly in cooperation with other systems. Usually, they are presented in the form of prescriptive statements to be enforced by the system-to-be. In the requirements elicitation phase besides understanding what problem needs to be solved and why we have to understand who are the entities involved and what are their responsibilities in solving the problem.

The system **specification** prescribes what the system does. It is a description of what happens at the interface between the machine and the world. A specification can be a formal model that describes what the system achieves in the world.

To bridge the gap between requirements and specification, we have to make **assumptions** on what environment property holds in the world. Assumptions are properties of the world, i.e. statements about expected behaviors of the environment. For example, we might take into account physical laws, network latency, user profiles, and so on.

In the design process, we first *model* requirements, specification, and environment assumptions with some formalism. Then, we have to *design* a system that meets its specifications. Finally, in the *analysis* phase, as system designers, we have to prove that the specification under the assumptions *satisfies* the requirements.

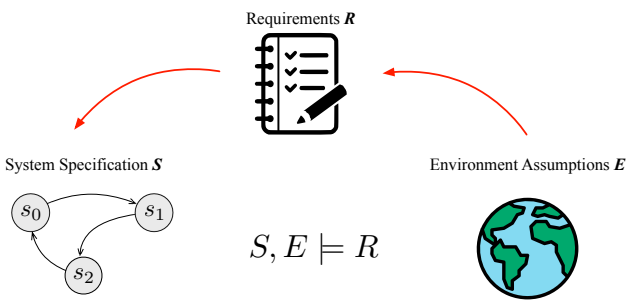


Figure 1.1: Interaction between system specificaiton, environment assumptions and requirments framed in a *trustworthiness argument*.

Let R, S, and E be the models of the requirements, specification, and environment assumptions, respectively. Then, we want to be able to prove that S under E satisfies R. In the context of this thesis, a system is *trustworthy* if $S, E \models R$, where $\models$ represents the 'entailment' operator. Figure 1.1 shows this relationship that we refer to as the trustworthiness argument. Such satisfiability relation assumes that:

— the requirements R precisely and unambiguously capture the problem to solve in the world by the system;

— the system behaves according to the specification S;

— the environment assumption E hold in the real world.

Any of the assumptions above can be violated. Such violation can be caused by the presence of uncertainty during the modeling or designing phase. In the following sections, we discuss the different sources of uncertainty that can pose several challenges to the system designer and ultimately can jeopardize the satisfaction of the trustworthiness argument. Specifically, we discuss the challenges affecting the formulation of requirements, system specification, and environment assumptions in Section 1.2. In Section 1.3, we give some background of the formal techniques used in this thesis to facilitate the satisfaction of the trustworthiness argument.

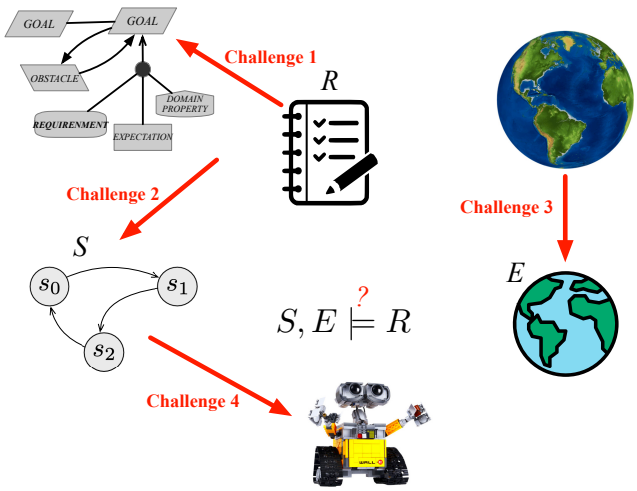


Figure 1.2: Challenges that could jeopardize the satisfaction of the *trustworthiness argument*.

1.2 Challenges and Related Work

In this section, we present some challenges that the system designer has to deal with while building a system that satisfies the trustworthiness argument. Figure 1.2 shows the different problems that can emerge and that can jeopardize the satisfaction of our trustworthiness argument. Challenge 1 concerns the elicitation and modeling of requirements while keeping them consistent and complete. Challenge 2 deals with the problem of producing a system specification from its requirements. Challenge 3 analyzes the gap between the environment assumptions made by the designer and the real world where the system is operating. Finally, challenge 4 deals with the gap between the system specification and the actual system that is being built. In the following sections, we describe all of these challenges and present some related work that deals with them.

1.2.1 Challenge 1: Capturing and modeling requirements

Requirement elicitation often revolves around human-related considerations that are intrinsically difficult to capture. Furthermore, existing requirement-management tools are mostly based on natural language constructs that leave space for ambiguities and conflicts. Goal models have been used over the years as an intuitive and effective means to capture the designer's intents and their hierarchical structure, mostly following an "optative" (i.e. expressing a *wish*) approach oriented to the design objectives.

Goal Models KAOS [24], TROPOS [25], and i* [26] are established formalisms, used to capture system goals and elicit high-level requirements. An extensive systematic mapping study in the area of goal-oriented requirement engineering has recently appeared [27].

A goal is a prescriptive statement of intent that the system should satisfy, formulated in a declarative way. Goals can be decomposed, progressing from high-level objectives to fine-grained system prescriptions [28]. In the context of this thesis, a *goal model* is a directed acyclic graph, where nodes denote goals. Child nodes can be linked to parent nodes via AND/OR refinement links. An AND-decomposed goal is satisfied if and only if all its sub-goals are satisfied. An OR-decomposed goal is satisfied if and only if at least one of its sub-goals is satisfied. OR-refinement links are used to represent different alternative options, usually concerning different systems.

Figure 1.3 shows an AND-refinement of a goal $\mathcal{G}$ into sub-goals where the leaf-nodes are assigned to be system *requirements*, environment *expectations* or *domain properties*. Domain properties are descriptive statements about the problem world (such as physical laws) while the environment expectations, or domain hypotheses, are only *assumed* to be true by the system designer.

Abstraction/Refinement in goal models The refinement process in goal models mostly follows informal procedures [29, 30]. Goal refinement can be performed during the elicitation process by posing *how* questions to find possible sub-goals (e.g., 'how can the goal $\mathcal{G}$ be satisfied?' or 'are there other sub-goals needed to satisfy $\mathcal{G}$?'). Goal abstraction uses, instead, *why* questions aiming to

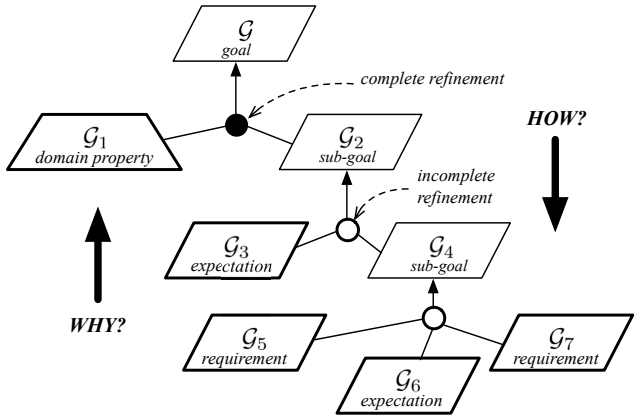


Figure 1.3: Goal model with AND-refinements and leaf nodes.

identify which goals may be established behind a set of elicited requirements. For example, in Figure 1.3, the parent goal of $\mathcal{G}_5$ can be inferred by asking 'why should $\mathcal{G}_5$ be satisfied by the system?' or 'is there any other parent goal that $\mathcal{G}_5$ contributes to?'

Consistency and Completeness Problems Establishing *correctness* of the refinement amounts to ensuring that the sub-goals are *consistent*, i.e., there are no *conflicts* among them, and *complete*, i.e., there are no behaviors left unspecified that could result in a violation of the high-level goal even if the lower-level goals are satisfied. We only refer to *internal completeness*, i.e., we are not concerned with investigating whether all the information required to define a design problem is in the specification [31]. Formally, we say that the refinement of goal $\mathcal{G}$ into sub-goals $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ is correct if and only if

$$\underbrace{\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \not\models \texttt{false}}_{\text{consistency}} \quad \wedge \quad \underbrace{\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \models \mathcal{G},}_{\text{completeness}}$$

where we denote by $\models$ the entailment operator between goals and say that $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ entails $\mathcal{G}$ to mean that, if all $\mathcal{G}_1, \ldots, \mathcal{G}_n$ are satisfied then $\mathcal{G}$ is satisfied. Similarly, we write $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \not\models \texttt{false}$ to indicate that the logical conjunction of $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\}$ does not lead to false.

1.2.2 Challenge 2: From requirement to specification

While requirements formulate what the system has to achieve in the world, the specification prescribes what the system does. A requirement is an informal description of the problem to be solved while a specification indicates the behaviors of the system in a formal language with precise semantics. Bridging the gap between requirement and specification is one of the challenges to be overcome by the system designer.

Use of formal (logic) languages Many results in the literature highlight the advantages of formulating system specifications in temporal logic language,

like linear temporal logic (LTL) or computation tree logic (CTL) [32–45]. Using formal languages makes behavioral specifications precise and unambiguous.

Expressing the system and its properties with formal languages like LTL is very useful to verify the correctness of the system via a model checker. We describe the syntax and semantic of LTL in Section 1.3.1 and the model checking problem in Section 1.3.2. However, logic formulas can be difficult to interpret for the end-user, and generating them can be an error-prone process [46–48]. Specification patterns have been proposed as a solution for bridging the gap between informal requirements and system specifications. In this thesis, we use general specification patterns proposed by Dywers [49] and in the context of robotic mission specifications we use the robotic patterns proposed by Menghi [48].

Specification Patterns with Scopes Dwyers et al. [49] developed a catalog of generic property specification patterns for a broader range of applications. Each pattern can be instantiated in a *scope*, which provides a way to define the extent to which a property must hold [49]. For example, for the *universality* pattern, in which we require that a property e be true, we can introduce the following scopes:

$$
\begin{aligned}
e \text{ global} &= \mathbf{G}(e) \\
e \text{ before } r &= \mathbf{F}(r) \rightarrow (e \ \mathbf{U} \ r) \\
e \text{ after } q &= \mathbf{G}(q \rightarrow \mathbf{G}(e)) \\
e \text{ between } q \text{ and } r &= \mathbf{G}((q \ \wedge \ \neg r \ \wedge \ \mathbf{F} \ r) \rightarrow (e \ \mathbf{U} \ r)) \\
e \text{ after } q \text{ until } r &= (\mathbf{G}(q \ \wedge \ \neg r \rightarrow ((e \ \mathbf{U} \ r) \mid \mathbf{G} \ p)))
\end{aligned}
$$

where q, r are also properties or events. The patterns proposed by Dwyers et al. [49] were also extended to incorporate time [50] and probability [51]. Autili et al. [52] present a unified catalogue of property specification patterns including, among others, the patterns mentioned above [49–51]. A description of the patterns in this catalogue [52] is also available online [53].

Robotic Missions Specification Robotic mission requirements are often ambiguous [54–56] and make it hard to assess the correctness of the specification [57–60]. In recent years, there have been many proposals for describing mission requirements based on: *i)* domain specific languages [61–63], *ii)* natural language [55], and *iii)* visual and end-user-oriented environments [64–67], mostly used for educational purposes. While the approaches above provide substantial contributions to the mission specification problem, solutions that can scale to complex missions and enable the deployment of service robots in everyday life are still elusive.

Robotic patterns have been proposed as a solution to recurrent mission specification problems based on the analysis of mission requirements in the robotic literature [48]. There are 22 patterns [48], capturing robot movements and actions performed as a robot move in the environment, organized into three groups: *core movement* patterns, *triggers*, and *avoidance* patterns.

For example, let us assume that the mission requirement is: 'A robot must patrol a set of locations in a certain strict order.' The designer can formulate

this requirement by using the *Strict Ordered Patroling* pattern, instantiated for the required set of locations. Let l_1, l_2, and l_3 be the atomic propositions of type *location* that the robot must visit in the given order. The mission requirement can then be reformulated as 'Given the locations l_1, l_2, and l_3, the robot should visit all the locations indefinitely and following a strict order,'[1] leading to the following LTL formulation:

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(l_1 \wedge \mathbf{F}(l_2 \wedge \mathbf{F}(l_3)))) \wedge (\neg l_2 \ \mathbf{U} \ l_1) \wedge (\neg l_3 \ \mathbf{U} \ l_2) \\
\wedge\ &\mathbf{G}(l_2 \rightarrow \mathbf{X}(\neg l_2 \ \mathbf{U} \ l_1)) \wedge \mathbf{G}(l_3 \rightarrow \mathbf{X}(\neg l_3 \ \mathbf{U} \ l_2)) \\
\wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\neg l_1 \ \mathbf{U} \ l_3)) \\
\wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\neg l_1 \ \mathbf{U} \ l_2)) \wedge \mathbf{G}(l_2 \rightarrow \mathbf{X}(\neg l_2 \ \mathbf{U} \ l_3)).
\end{aligned}$$

As shown in this example, a robotic pattern can significantly facilitate the difficult and error-prone task of mission specification.

1.2.3 Challenge 3: Environment assumptions are unknown

Any autonomous system acts on an environment, however, to prove the trustworthiness argument we must produce a *model* of the environment. The model represents an abstraction of the actual environment in which the system operates. The correctness of the trustworthiness argument is subject to the correctness of the model abstracting the environment. If the actual environment does not match the model then we cannot guarantee that the system requirements are respected.

We differentiate two kinds of assumption violations. On one hand, we have the assumptions of *internal* system components requirements that are in conflict; these violations can be detected and mitigated at design-time. On the other hand, we have assumptions that are *external* to the system being built, and their violation can only be detected at run-time.

Incomplete Refinements at Design-Time When eliciting the top-level requirements of a specification in a hierarchical way, we may discover additional assumptions, associated with lower-level requirements in the hierarchy, which were not known *a priori*. This inconsistency between assumptions at different levels of the hierarchy may be a reason for incompleteness in the refinement, which results in the assumptions made on top-level requirements being violated.

Often, initially, the designer does not have a clear idea of what the assumptions of the top-level requirements should be. The assumptions required by the implementation might conflict with the assumptions originally made by the designer since they were agnostic on the components being used to realize the requirements. We might then have a series of iteration where assumptions are *negotiated* and *refined* to accommodate different system implementations.

World Assumptions Determined at Run-Time Autonomous Systems can operate under highly complex and uncertain environments. One of the challenges is modeling the environment in which the autonomous system operates and keeping the model updated as the environment changes. Sometimes

[1] `http://roboticpatterns.com/pattern/strictorderedpatrolling/`

the environment model falling outside its abstraction can only be captured at runtime. One of the reasons for inaccuracy in the model is that sometimes some of its assumptions are not held at runtime [68]. For example, making simplifying assumptions on the *transmission delay* of network packets might result in the assumption violation when the system is running due to network congestion.

Sometimes the environment model falling outside its abstraction can only be captured at runtime. One of the reasons for inaccuracy in the model is that sometimes some of its assumptions are not held at runtime [68]. For example, making simplifying assumptions on the *transmission delay* of network packets might cause the assumption violation when the system is running due to network congestion.

Typically, the environment model, sometimes referred to as *world model*, is a structure created manually, which can be monitored and updated at runtime [69]. Ghosh et al. show how to algorithmically repair specifications that include environment assumptions [70, 71]. Li et al. [72] propose an approach with the human in the loop where, as soon as the environment fails to meet its assumptions, a flag is raised and the control of the system is safely switched to the human operator.

In the context of reactive synthesis, we have that incomplete or wrongly specified environment assumptions are a common reason for the unrealizability of a system specification. Several works go in the direction of finding new assumptions or relaxing existing ones in a way that the system becomes realizable [73–77].

1.2.4 Challenge 4: Gap between system specification and system implementation

The system specification describes the function that the system should provide (i.e. *what*) while the system implementations provide the components and their interconnection needed to realize the function (i.e. *how*). Verifying that the *specification* is compliant with some requirements might not be enough to certify the *system* is also compliant. It can happen that the behaviors that the system can exhibit might not be fully captured by its specification, hence leading to the *gap* between specification and implementation that jeopardizes the trustworthy argument. Formal methods such as *reactive synthesis* can automatically generate a system implementation that is mathematically proven to be compliant with its specification. Other formal techniques such as *run-time verification* can monitor the system execution and detect when its behaviors violate some specification. We take a look at these techniques in Section 1.3.4 and Section 1.3.3.

More generally, *Platform-based design* [78] is a paradigm that capture the design process of *mapping* a high-level specification to a potential implementation via a sequence of top-down refinement step that meet a bottom-up abstractions. At the top, we have the *application space* with the system specification, at the bottom we have the *architectural space* with collections of system components. The system specification is an instance of the application space which via a series of *refinement steps* is eventually *mapped* on an instance of the architectural space, i.e. a composition of system components according to some

composition rule. The system specification constrains the space of possible system architecture that is possible out of all the available ones. Once an architecture is found at some abstraction level, it then becomes the *specification* at the lower abstraction level. Another architectural space is then searched and the process is repeated until all the components are implemented in their final form. Each abstraction layer is defined by a *platform*.

1.3 Trustworthiness

To address the challenges described in Section 1.2 and provide guarantees of correctness in establishing the trustworthiness argument, we need to establish formally:

1. what are system and specification,
2. what does it mean for requirements to be consistent and complete,
3. what are system components and how they can be aggregated,
4. *refinement/abstraction* relationships among specification and components,
5. what does it mean that aggregation of components *implements* a specification,
6. how does a system learn and adapts to the environment with *reinforcement learning*.

In this section, we present the formalisms used in this thesis that can be used to address all the points above and establish trustworthiness in the system.

1.3.1 Linear Temporal Logic

The system can be modeled as a set of behaviors over some variables.

Definition 1.3.1 (System)**.** A system Σ is a set of variables V over a domain D in which the variables take value, and a set of *behaviors* over V.

Definition 1.3.2 (State)**.** A state s of a system Σ is a *valuation* (i.e. an assignment of a value) of all its variables V within their domain D. We denote with S the set of all states of a system Σ.

Definition 1.3.3 (Behavior)**.** A *behavior* of a system Σ is an infinite sequence of states $s_0, s_1, s_2, ...$ in S.

Definition 1.3.4 (Language)**.** A *language* of a system Σ, written $\mathcal{L}(\Sigma)$ is the set of all behaviors.

Definition 1.3.5 (Atomic Proposition)**.** An atomic proposition is a statement on system variables that has a unique truth value (i.e. it can be either true or false). Given an atomic proposition π and a state of the system $s \in S$ we can always determine if π *holds* in s (i.e. *pi* is true at the state s and we write $\pi \models s$) or otherwise (i.e. $\pi \not\models s$).

We model the system as a *transition system*, i.e. a having a set of *states* (static structure) and *transitions* (dynamic structure). One of the way to represent a transition system is a Kripke structure, which is formally defined as follows:

Definition 1.3.6 (Kripke structure)**.** $\Sigma = (S, S_0, \delta, L)$ where:

— S is a set of states.

— S_0 is a set of *initial* states.

— $\delta \subseteq S \times S$ is a transition relation which is *left-total*, i.e. for every state $s \in S$ there has to be at least one other state s' state such that $(s, s') \in \delta$.

— $L : S \to 2^{AP}$ is a labeling (or *interpretation*) function that maps a state to a set of atomic propositions (AP). We can think of L as assignment of truth values to all the atomic propositions that depends on the state of the system. That is $L(s)$ contains all the atomic propositions which are `true` in the state s.

Definition 1.3.7 (Run)**.** A *run*, (also referred as *path* or *execution*), of a system Σ defined as a Kripke structure (S, S_0, δ, L) is an infinite sequence of states $s_0, s_1, s_2, ...$ in S such that:

1. $s_0 \in S_0$
2. $\forall i \geq 1, \delta(s_{i-1}, s_0)$

Definition 1.3.8 (Trace)**.** A *trace*, of a system Σ defined as a Kripke structure (S, S_0, δ, L) is a sequence of labels observed during a *run* of Σ, i.e. $L(s_0), L(s_1), L(s_2)...$.

Definition 1.3.9 (Property)**.** A *property* φ is a set of runes. A run σ *satisfies* a property φ if $\sigma \in \varphi$. A run σ *violates* a property φ if $\sigma \notin \varphi$.

Linear Temporal Logic (LTL) is one of the logic that can be used to specifying *properties* of a system related to its evolution over time. LTL can express the properties of the system over a single timeline. We can construct LTL formulas over AP according to the following recursive grammar:

$$\varphi := p \in AP \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\ \varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$$

where φ, φ_1, and φ_2 are LTL formulas. From the negation ($\neg$) and disjunction ($\vee$) of the formula we can define the conjunction ($\wedge$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). Boolean constants *true* and *false* are defined as $true = \varphi \vee \neg\varphi$ and $false = \neg true$. The temporal operator **X** stands for *next* and **U** for *until*. Other temporal operators such as *globally* (**G**) and *eventually* (**F**) can be derived as follows: $\mathbf{F}\ \varphi = true\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi = \neg(\mathbf{F}(\neg\varphi))$.

Let $\sigma = s_0, s_1, s_2, ...$ be a *path* of the system Σ. We denote with σ^i the *suffix* of σ starting at s_i, e.g. $\sigma^3 = s_3, s_4, s_5, ...$ Whether σ *satisfies* and LTL formula φ is defined by the satisfaction relation $\models$ as follows:

— $\sigma \models$ `true`

— $\sigma \not\models$ `false`

— $\sigma \models p$ iff $p \in L(s_0)$

— $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$

— $\sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$

— $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$

— $\sigma \models \varphi_1 \rightarrow \varphi_2$ iff $\sigma \models \varphi_2$ whenever $\sigma \models \varphi_1$

— $\sigma \models \mathbf{X}\varphi$ iff $\sigma^1 \models \varphi$

— $\sigma \models \mathbf{G}\varphi$ iff, for all $i \geq 0$, $\sigma^i \models \varphi$

— $\sigma \models \mathbf{F}\varphi$ iff, there some $i \geq 0$ such that $\sigma^i \models \varphi$

— $\sigma \models \varphi_i \mathbf{U} \varphi_2$ iff, there some $i \geq 0$ such that $\sigma^i \models \varphi_2$ and for all $j = 0, ..., i-1$ we have $\sigma^j \models \varphi_1$

1.3.2 Model Checking

Model checking is an algorithmic method for establishing whether a system satisfies a specification formalized as a temporal logic formula.

Definition 1.3.10 (Model Checking Problem)**.** Given a system Σ and an LTL formula φ the model checking algorithm returns weather Σ *satisfies* φ. If *"yes"*, written $\Sigma \models \varphi$, then we have a proof of the system correctness with respect to the property φ. If *"no"*, written $\Sigma \not\models \varphi$, typically we receive a *counterexample*, i.e. a *trace* of the system where φ does not hold.

The model-checking problem reduces in checking whether all the possible behaviors of the system is a subset of all possible executions of the property. That is whether the language of the system is a subset of the language of the property, i.e. $\mathcal{L}(\Sigma) \subseteq \mathcal{L}(\varphi)$.

1.3.3 Runtime Verification

Runtime verification is only concerned with the *detection* of violations (or satisfactions) of properties on a system. While model checking considers *all executions* of a given system to assert if it satisfies a given property, runtime verification only checks certain finite execution traces.

Definition 1.3.11 (Runtime Verification)**.** A verification technique that allows checking whether a *run* of a system satisfies or violates a given property.

While model checking checks if the language of a system is included in the language of the property, runtime verification answer whether a given *word* is included in some language. Checking whether a finite system execution trace satisfies a property is performed using a *monitor*.

Definition 1.3.12 (Monitor)**.** A *monitor* is some device that reads a finite trace w and returns weather it *satisfies* a given correctness property φ.

1.3.4 Reactive Synthesis

An alternative technique to model checking and run-time verification is to systematically *build* the system which satisfies a given specification where its correctness is guaranteed from its construction.

Definition 1.3.13 (Reactive Synthesis)**.** Given an LTL specification φ over a set of atomic proposition partitioned into inputs and outputs, i.e. $AP = \mathcal{I} \cup \mathcal{O}$, the *synthesis problem* determines whether there exists a finite-state machine $\mathcal{M} = (S, \mathcal{I}, \mathcal{O}, s_0, \delta)$ that satisfies φ. Where S is the set of states, $s_0 \in S$ is the initial state, and $\delta : S \times 2^{\mathcal{I}} \to S \times 2^{\mathcal{O}}$ is the transition function. $\mathcal{M}$ satisfies a formula ϕ if all its runs satisfy ϕ. If such machine exists it computes it and we say that $\mathcal{M}$ *realizes* the formula φ.

1.3.5 Contracts for System Design

We use assume/guarantees contracts to model the system specification for the system components. A/G contracts explicitly distinguish the responsibilities of a component (guarantees) from those of its environment (assumptions). In the following, we provide informal definitions of the different operations and the formulae used to compute them. We refer to the literature [13] for the formal definitions.

Definition 1.3.14 (Components)**.** System components are typically regarded as *open* systems [79], i.e. systems that perceive some inputs from other components in the system or the external world and it produces some outputs. Anything external to the component is referred to as the *environment*. Components represent design elements that express a *behavior* over a set of inputs/output *variables* which are defined over a certain *domain*. Formally, we can regard a component in the same way we have defined a system in 1.3.1 where the system variables V are partitioned into input and output.

Definition 1.3.15 (Contract)**.** A *contract* $\mathcal{C}$ is a triple (V, A, G) where V is a set of system *variables* (including, e.g., input and output variables or ports), and A and G are sets of *behaviors* over V. For simplicity, whenever possible, we drop V from the definition and refer to contracts as pairs of assumptions and guarantees, i.e., $\mathcal{C} = (A, G)$.

The assumptions A express the behaviors that are expected from the environment, while the guarantees G express the behaviors that a component promises under the environment assumptions. In this thesis, we express assumptions and guarantees as sets of behaviors satisfying a logic formula; we then use the formula itself to denote them, with a slight abuse of notation, whenever there is no confusion.

Definition 1.3.16 (Contract Environment)**.** An environment E satisfies a contract $\mathcal{C}$ whenever E and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of E are included in the assumptions of $\mathcal{C}$, i.e., when $|E| \subseteq A$, where $|E|$ is the set of behaviors of E.

Definition 1.3.17 (Contract Implementation)**.** A component M, or *implementation*, satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables, and all the behaviors of M are included in the guarantees of $\mathcal{C}$ when considered in the context of the assumptions A, i.e., when $|M| \cap A \subseteq G$, where $|M|$ represents the set of the behaviors of M. We refer to the set of *maximal implementations* of $\mathcal{C}$ as $M_{\mathcal{C}} = G \cup \overline{A}$, where $\overline{A}$ denotes the complement of

A. That is, $M_{\mathcal{C}}$ is the maximal set of behaviors (with respect to set inclusion) satisfying the contract $\mathcal{C}$.

Definition 1.3.18 (Contract Equivalence)**.** Two contracts $\mathcal{C}_1 = (A, G_1)$ and $\mathcal{C}_2 = (A, G_2)$ are semantically equivalent if they possess identical variables, identical assumptions, and such that $G_2 \cup \overline{A} = G_1 \cup \overline{A}$, i.e. they have identical maximal implementations $M_{\mathcal{C}_1} = M_{\mathcal{C}_2}$.

Definition 1.3.19 (Contract Saturation)**.** A contract $\mathcal{C} = (A, G)$ can be placed in saturated form by re-defining its guarantees as $G_{sat} = G \cup \overline{A}$. A saturated contract explicitly contains the biggest set of guarantees possible, i.e. the guarantees coincide with the union of the behaviors of all its implementations.

A contract and its saturated forms are semantically equivalent since the two contracts possess identical sets of environments and implementations. Therefore, in the rest of the thesis, we assume that all the contracts are expressed in saturated form.

Definition 1.3.20 (Compatibility and Consistency)**.** A contract $\mathcal{C}$ is *compatible* if there exists an environment for it, i.e., if and only if $A \neq \emptyset$. Similarly, a saturated contract $\mathcal{C}$ is *consistent* if and only if there exists an implementation satisfying it, i.e., if and only if $G \neq \emptyset$. We say that a contract is *well-formed* if and only if it is compatible and consistent. We detail below the contract operations and relations used in this thesis.

Refinement establishes a pre-order between contracts, which formalizes the notion of replacement.

Definition 1.3.21 (Contract Refinement)**.** Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. $\mathcal{C}$ refines $\mathcal{C}'$, denoted by $\mathcal{C} \preceq \mathcal{C}'$, if and only if all the assumptions of $\mathcal{C}'$ are contained in the assumptions of $\mathcal{C}$ and all the guarantees of $\mathcal{C}$ are included in the guarantees of $\mathcal{C}'$, that is, if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement entails relaxing the assumptions and strengthening the guarantees. When $\mathcal{C} \preceq \mathcal{C}'$, we also say that $\mathcal{C}'$ is an *abstraction* of $\mathcal{C}$ and can be replaced by $\mathcal{C}$ in the design.

Contracts associated with distinct implementations can be combined via the composition operation ($\|$) to specify the composition between the corresponding implementations.

Definition 1.3.22 (Contract Composition)**.** Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. The composition $\mathcal{C} = (A, G) = \mathcal{C}_1 \parallel \mathcal{C}_2$ can be computed as follows:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, \tag{1.1}$$

$$G = G_1 \cap G_2. \tag{1.2}$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, hence the operation of intersection in (1.2). An environment for $\mathcal{C}$ should also satisfy all the assumptions, motivating the conjunction of A_1 and A_2 in (1.1). However, part of the assumptions in $\mathcal{C}_1$ may be already supported by $\mathcal{C}_2$ and *vice versa*. This allows relaxing $A_1 \cap A_2$ with the complement of the guarantees of $\mathcal{C}$ [13].

Different contracts on a single implementation can be combined using the conjunction operation ($\wedge$).

Definition 1.3.23 (Contract Conjunction)**.** Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. We can compute their conjunction by taking the greatest lower bound of $\mathcal{C}_1$ and $\mathcal{C}_2$ with respect to the renement relation. Intuitively, the conjunction $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ is the weakest (most general) contract that refines both $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}$ can be computed by taking the intersection of the guarantees and the union of the assumptions, that is:

$$\mathcal{C} = (A_1 \cup A_2, G_1 \cap G_2).$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, while being able to operate in either of the environments of $\mathcal{C}_1$ or $\mathcal{C}_2$.

1.3.6 Reinforcement Learning

Reinforcement Learning (RL) [80] is a machine learning approach where a software agent learns *what to do* by taking actions on an environment in order to maximize a numerical reward signal. Actions may not have an immediate effect on the reward, instead, rewards might be *delayed*. The mathematical model typically used to formalize the reinforcement learning problem is the *Markov decision process*.

Definition 1.3.24 (Markov Decision Process (MDP))**.** An MDP is a tuple $< S, A, T, R, \gamma >$ where:

- S is a finite set of states
- A is a finite set of actions
- $T : S \times A \times S \to [0, 1]$ is a state transition probability function such that for any $s, s' \in S$ and any action $a \in A$ allowed in state s, $T(s, a, s')$ returns the probability associated to the transition from state s to state s' when performing action a.
- $R : S \times A \times S \to \mathbb{R}$ is a reward function such that $R(s, a, s')$ is the reward returned by taking action a from state s and reaching the state s'.
- $\gamma \in [0, 1]$ is the *discount factor* which indicates the preference between receiving a short-term reward ($\gamma = 0$) versus a long-term reward ($\gamma = 1$).

The agent and the environment interact in discrete time steps as shown in the Figure 1.4. At each time step $t = 0, 1, 2, ...$ the agent receives observations from the environment that correspond to a certain state $s_t \in S$. It then chooses an action $a_t \in A$ among the set of possible actions. The action is then applied to the environment that moves to a state s_{t+1} and returns a reward r_{t+1} to the agent. At each step all future rewards are discounted by γ as follows: $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ...$ The discount factor represents the difference in importance between future rewards and present rewards.

Definition 1.3.25 (Markov Property)**.** The MDP satisfies the *Markov Property*: the next state only depends on the current state and the chosen action, so it is conditionally independent of all previous states and actions. We have that $T(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}...) = T(s_{t+1}|s_t, a_t)$.

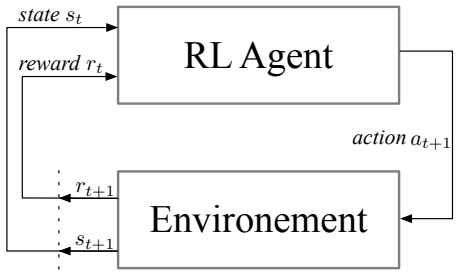


Figure 1.4: Reinforcement learning framework showing the interaction between the agent and the environment.

Definition 1.3.26 (Return)**.** The *return* $G = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$ is the sum of all the *discounted* rewards that the agent will get from time step 0 infinitely into the future.

Definition 1.3.27 (Policy)**.** A policy π fully defines the behavior of a RL agent. It can be either deterministic or stochastic. If the policy is deterministic it simply mapping function from state to action $\pi : S \to A$. A stochastic policy returns a probability distribution over actions given a state: $\pi : S \times A \to [0, 1]$. That is, it maps each state s and action a to the probability that action a is taken in state s, where we have that $\sum_{a \in A} \pi(s, a) = 1$ for any $s \in S$.

Definition 1.3.28 (State Value)**.** $V_\pi : S \to \mathbb{R}$ returns the *expected return* received by the agent when starting from state s and following a policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right] \tag{1.3}$$

Since the reward function $R(s, a, s')$ depends on the current state s, the action a and the next state s' in which the agent *lands*, which depends on the transition probability function $T(s, a, s')$, we need to sum of all its possible reward values weighted by their probability of being observed. With $\mathbb{E}_\pi[G_t | s_t = s]$ we denote the *expected value* of the return at time-step t given that the agent is in the state s and follows a policy π.

Definition 1.3.29 (Action Value)**.** $Q_\pi : S \times A \to \mathbb{R}$ returns the value of taking action a in state s under a policy policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right] \tag{1.4}$$

Definition 1.3.30 (Solving the MDP problem)**.** Given an MDP, find the *optimal policy* π^*, that is the policy having *optimal state-value function* V^* for all $s \in S$ and *optimal action-value function* Q^* for all $s \in S$ and $a \in A$, defined as:

$$V^*(s) = \max_\pi V_\pi(s) \tag{1.5}$$

$$Q^*(s) = \max_\pi Q_\pi(s, a) \tag{1.6}$$

Definition 1.3.31 (Bellman optimally equations)**.** Specify optimally in a *recursive way.* Intuitively the *state value* of a state under optimal policy must be equal to the *action value* for the best action form that state

$$V^*(s) = \max_{a \in A} Q^*(s) \tag{1.7}$$

Where we have that the optimal action value for taking action a in a given state s is equal to the instantaneous reward for the transition plus the optimal value of state s' (discounted), for all outcomes s' and their probability of occurrence. That is:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{1.8}$$

Substituting equation (1.8) in equation (1.7) we have that:

$$V^*(s) = \max_{a \in A} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{1.9}$$

If the elements of the MDP are fully known, then the problem of solving the MDP becomes a *planning* problem which can be solved via dynamic programming algorithms. Iterative methods such as *value iteration* can solve the Bellman equations and compute the optimal policy.

However, usually, we do not have access to the full MDP, instead, the software agent interacts with the environment collecting *samples.* A sample is a tuple (s, a, s, r) representing the experience of the agent performing an action a in a state s, landing in a state s' and receiving an immediate reward r. In *model-based* learning, we use the sample to estimate the underline MDP and then do planning on it. In *model-free* learning, we use the sample to directly estimate the values of the states.

For example, Q-learning [81] is a simple and effective way to learn the optimal policy by interacting with the environment via samples. It estimates the action-values for each state, i.e. the long-term expected return of each action that can be executed from a given state.

Definition 1.3.32 (Q-Learning update)**.** In Q-learning at each interaction with the environment at time step t we update *q-value* associated with the state s and action a by integrating the *reward* received for performing action a in state s and landing in state s'.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1}, \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.10}$$

where $\alpha \in [0, 1]$ is the *learning rate* and determines how much the newly acquired information from the sample overrides the previous knowledge of the action-value.

1.4 Goals and Methodology

The overall goals of this thesis are:

G1: Analyze system requirements and translate them in system specifications also in presence of uncertainty.

G2: Design and build Autonomous Systems that satisfy the requirements while maximizing some system quality.

The goal **G1** aims at modeling system requirements and transferring the designer's intentions into some system specification. The uncertainty can be because the designer does not know *how* the system should achieve some goal or *what* component the system should be composed of. For the first goal we have formulated the following research questions:

RQ1.1: How can we formulate system specifications from system requirements also in presence of uncertainties?

RQ1.2: How do we make sure that system requirements are complete and consistent and do not express unwanted behaviors?

The goal **G2** aims at engineering the actual system and prove its compliance with some specifications while minimizing some cost function (e.g. reuse of system components). For the second goal we have formulated the following research questions:

RQ2.1: How to make sure that the resulting system satisfies certain desired properties while maximizing some quality?

RQ2.2: How to provide assurances when several autonomous systems collaborate to create a new and more complex system?

Research question **RQ1.1** aims at producing a formal system specification from informal designer intentions.

In cases, of uncertainty on *how* the system should achieve some requirements. We have used specification patterns and robotic specification patterns to formulate an unambiguous specification. We have also refined the specification via several refinement steps both performed manually and automatically from a collection of pre-defined goals. This approach is found in Paper D, Paper E, and Paper F.

In cases of uncertainty on *what* the system should do to achieve its goals, instead of breaking down its requirements via refinement steps, we have formulated them as a reward function and used reinforcement learning. Given the reward function, the system figures out by trial-and-error what it should do to maximize the cumulative sum of rewards. We have used this approach in Paper C and Paper B.

The question **RQ1.2** builds in the direction of assuring that the goals as intended by the designer are complete, consistent, and correctly transferred to the AS itself.

In Paper B, we deal with the problem of conveying the functional requirements to a machine learning agent solely using the reward function. A problem with such systems is that the agent could manifest unwanted behaviors and consequences. We aim to give the system designer more control and understanding of the reward function on which the reinforcement learning agent is going to build its decision-making policy.

In Paper D, Paper E, and Paper F, we use Contract-Based Design as a methodology to guide the design of the system. First, we formalized the requirements in specifications using patterns, and the specifications in assume-guarantee contracts. Then, we can reformulate questions on completeness and consistency of system requirements into question checking refinement, compatibility, and consistency on assume-guarantees contracts.

The question **RQ2.1** aims at producing a system that satisfies its specification in all environmental conditions while minimizing some cost.

In Paper D, Paper E, and Paper F we generate the final system acting on the environment via *reactive synthesis*, hence having the guarantees that the specification will never be violated. We also maximize the *reuse* of existing system components by automatically refining the system specification from a collection of pre-defined goals. In Paper A we built a model of the system using *timed-automata* and automatically check its compliance with some important properties of its specification using model checking. Finally in Paper C we formally *verify* that the behavior emerging from the system is compliant with some important safety properties (*invariants*) also while the system is learning and exploring the environment using runtime monitoring techniques.

The research question **RQ2.3** does not focus on the behavior of a single autonomous system but instead, deals with a system constituted by several individual systems (System of Systems, SoS). We want to investigate how current verification techniques can be applied to such systems. We have addressed this research question in Paper A by modeling a SoS scenario where multiple vehicles have to interact to form a platoon. The individual systems can independently adapt their behavior at runtime and the rules defining the SoS are modeled at design time. We have modeled different modes where the SoS can be and defined properties we want the SoS to have as *invariants*. Finally, we have formally verified that in randomly generated scenarios the resulting system is compliant with them using model checking.

Methodology We have used *Design Science* as research methodology [82]. The two main activities are of design science are: (i) designing an artifact, (ii) empirically evaluating its performance in a context. In most of the papers addressing the artifact is the proposed method. In Paper C and Paper B, we have implemented a framework that supports the proposed methods; the validation has been done by defining a case study and collecting data from experiments conducted by *computer simulation* [83]. In Paper E, Paper D, and Paper F we have also implemented the framework and validated the methodology both by collecting data from experiments and mathematically proving the correctness of some results. In Paper A the artifact is the protocol for vehicle platooning and it has been validated by formal-verification (model-checking) on randomly generated scenarios.

1.5 Summary of Contributions

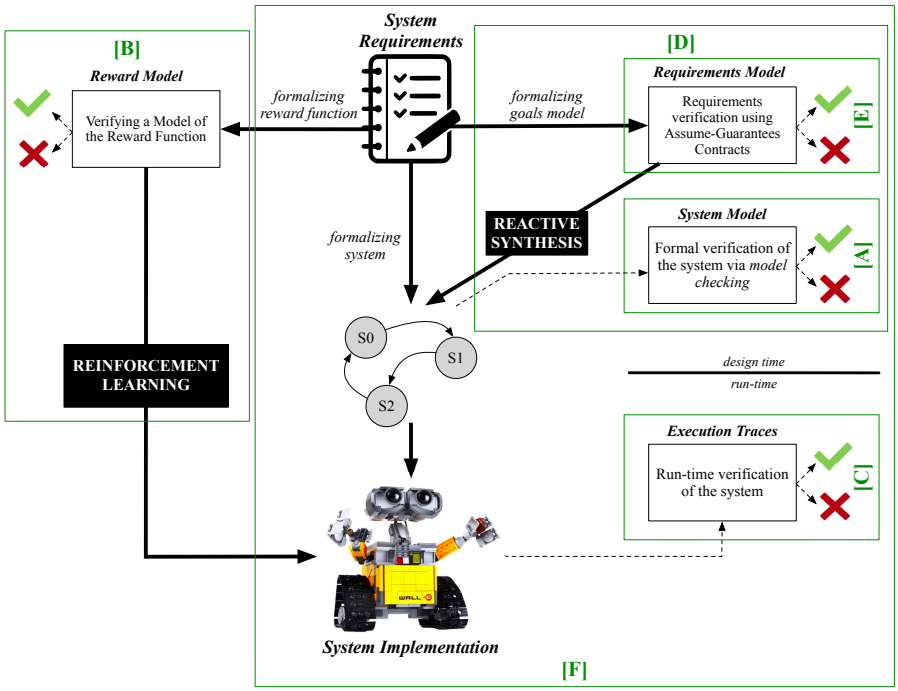


Figure 1.5: Overall contribution. The letters indicate the papers contributing in different parts of the framework.

Figure 1.5 maps the contributions of this thesis into a general framework describing the design process of building a trustworthy autonomous system. Starting from an informal description of the system requirements, we have expressed them in terms of a reward function (Paper B) or in terms of goal model formalized with assume-guarantee contracts (Paper E). A model of the system can be automatically generated from the assume-guarantee contracts via *reactive synthesis* (Paper D). Otherwise, as in Paper A, we have also directly modeled the system and then checked its compliance with some properties via model checking. When the model of the system is not available for a formal verification at design-time we have analyzed the trace of the system at run-time (Paper C). Finally, in Paper F we propose an all-encompassing framework to model requirements and realize provably correct system implementations using a combination of reactive synthesis and run-time monitoring. In the following sections, we describe the included papers.

1.5.1 Building a system model and verify invariants

Some system requirements can be expressed in terms of *invariants*, meaning properties that always have to hold, despite the system adaptations. Such

properties can target an individual system of multiple systems and the way they interact with each other. Autonomous Systems can also collaborate with other systems forming a System of System (SoS). Due to the complexity of such systems, it becomes hard to verify the correctness of their actions.

Formal verification techniques such as model checking can prove that a system satisfies certain desired formal properties. In **Paper A**, we model an Autonomous Systems as a network *timed-automata* and verify that certain properties hold when they interact. Specifically, we have modeled invariants in terms of temporal logic properties and formally verified such invariants in a system of systems scenario. We have successfully verified several invariants on a vehicle platooning protocol with randomly generated scenarios.

1.5.2 From the system requirements to reward function

An intelligent autonomous system must figure out how to achieve its goals by itself. The job of a system designer is to specify *what* goal to achieve and the decision-making component of an Autonomous System must figure out *how* to achieve it. In reinforcement learning, the only way that the system designer has to convey the goal to the agent is through the *reward function*. The encoding of system goals into a reward function can lead to unexpected behaviors in the agent, either because the designer does not include or have the correct information or because she/he makes mistakes during the design of the reward function.

Paper B addresses some of the challenges identified in [84] regarding the transfer of *goals* to a reinforcement learning agent, as also shown in Figure 1.5. We have proposed an approach for engineering complex reward functions that can be formally verified against defined properties at design time and automatically enforced to the agent at runtime. We aim to reduce the gap between the designer's intention and the reward specification. By embedding more domain knowledge in the reward function one could avoid the reward hacking phenomenon; this would also help the agent to learn the desired policy faster [85]. However, as reward functions become more complex, in turn, it becomes harder to spot mistakes and to be confident whether the reward values that are finally sent to the agent actually reflect the designer's intentions.

Our work goes in the direction of a better structuring of the reward function with the aim of closing the gap between the designer informal goals and the reward signal. Our contribution is the design and validation of a software infrastructure that enables the verification and enforcement of reward functions to an RL agent. From a high-level perspective our approach, which we have called MoVEMo, consists of four steps:

1. *Modeling* complex reward functions as a network of state machines.

2. *Formally verifying* the correctness of the reward model.

3. *Enforcing* the reward model to the agent at runtime using a monitoring and enforcing approach called Larva.

4. *Monitoring* the behavior of the agent as it transverses the state machines to collect the rewards.

Steps 1 and 2 are performed by the designer that iterates the reward function model until it is compliant with the high-level properties that she/he expresses. Steps 3 and 4 are automatically derived and performed from the reward model. We have validated our approach in the context of self-driving cars with an open-source driving simulator.

1.5.3 Train a system and later monitor its invariants

Using techniques such as *reinforcement learning* we can create systems that autonomously learn which action to execute in order to achieve the desired *informal* goal. When a change occurs, machine learning techniques allow the system to autonomously learn new policies and strategies for actions execution. This flexibility comes at a cost: the developer has no longer full control over the system behavior. To overcome this issue, we believe that machine learning techniques should be combined with suitable reasoning mechanisms aimed at assuring that the decisions taken by the machine learning algorithm do not violate safety-critical requirements.

Paper C describes how to combine the decision-making agent with the assurance of safety-critical properties. The approach aims at creating systems that, on one hand, are able to learn and adapt their behavior based on changes that occur in the environment using *reinforcement learning*, on the other are able to ensure that adaptation does not cause invariants violation using *runtime monitoring*. The runtime monitoring part is explained in detail and evaluated in Paper C.

Our approach uses reinforcement learning in combination with runtime monitoring to prevent the agent from performing catastrophic actions in the environment. The approach is general and external to the RL algorithm, so it does not modify how the RL algorithm works; in this sense, the approach is agnostic to the RL algorithm since one could use any RL algorithm. WiseML wraps the RL agent at its interfaces and it places it inside a *safety envelope* that protects it from performing actions that violate its safety-critical requirements.

The requirements of the RL agent are expressed in terms of *patterns*, they describe the safety properties to be enforced by the monitoring component. We have implemented four patterns that one can use to model the invariants: *absence*, *globally*, *precedence* and *response*. Our results show that the runtime monitors will always prevent the agent from violating any of the modeled properties. Furthermore, the RL agent will converge faster to its goals thanks to *reward shaping* that steer the agent towards its goal by modifying its rewards at runtime, according to the compliance or the violation of the monitored properties.

1.5.4 Modeling requirements as Goal Model and incrementally refining them

Goal models have been used over the years as an intuitive and effective means to capture the designer's objectives and their hierarchical structure, mostly following an optative approach (i.e. expressing a *wish*) oriented to the design objectives. Contracts, on the other hand, enable formal requirement analysis in a modular way, rooted in sound representations of the system semantics and

decomposition architecture, and offering an indicative approach oriented to the system components and their interactions, which are less explicit in goal models.

In PAPER E we present a framework, COGOMO (Contract-based Goal Modeling), for systematic requirement analysis, which leverages a new formal model, termed contract-based goal tree, to represent goal models in terms of hierarchies of contracts. Contract operations and relations provide formal support for notions such as stepwise *refinement* of high-level contracts into lower-level contracts, *compositional reasoning* about contract aggregations, and *reuse* of pre-designed components satisfying a contract. COGOMO addresses correctness and completeness of goal models by formulating and solving contract consistency and refinement checking problems. Specifically, the contributions of the paper can be summarized as follows:

— A novel formal model, namely, *contract-based goal tree (CGT)*, which represents a goal model as a hierarchy of assume-guarantee (A/G) contracts.

— Algorithms that exploit the CGT as well as contract-based operations to detect conflicts and perform complete hierarchical refinements of goals. Specifically, we introduce mechanisms that help resolve inconsistencies between goals during refinement and a *goal extension* algorithm to automatically refine the CGT using new goals from a library.

— A tool, which implements the proposed model and algorithms to incrementally formalize and refine goals via an easy-to-use web interface.

We illustrate the effectiveness of our approach and supporting tool on a case study motivated by vehicle platooning.

1.5.5 Modeling system specifications and automatically realize them in the context of robotic missions

The advent of multipurpose service robots, required to accomplish various domain-specific missions, calls for new languages and tools to enable end-users to accurately specify complex missions.

In PAPER D, we propose a framework, named CROME (Contract-based RObotic Mission spEcification), that explicitly addresses the problems of *specification reuse* and *environment modeling* in mission specification, enabling the designer to cope with the variability of the application scenarios of a robotic mission. By building on recent work on contract-based requirement engineering, leveraging *context-aware contract models* and patterns to generate controller specifications, we decouple the task specification from the specification of the context in which the task is executed. End users explicitly specify the various mission tasks together with their contexts. The overall mission is then automatically compiled by CROME. CROME contributes to the following aspects of the mission specification process:

— *Formulating mission requirements.* We model each requirement as a *goal*, expressed using a set of previously proposed patterns [48, 49]. Goal models have been used over the years as an intuitive and effective means to capture the designer's objectives and their hierarchical structure [86].

In this work, we augment the notion of goal to explicitly include a concept of *context*, which enables building mission specifications that are adaptable to different environmental conditions. Contexts help capture the variability associated with a mission goal so that the same goal can be implemented in different ways when used in different contexts.

— *Generating mission specifications.* We introduce a novel model, termed *contract-based goal graph* (CGG), which is automatically generated to formalize a mission and its sub-missions. The CGG is a graph of goals where the root node represents the overall mission, its immediate children represent mission *scenarios*, and the rest of the nodes are part of the sub-missions. In a CGG, goals are captured by assume-guarantee contracts [13] and are linked together using operations and relations between contracts. We differentiate the scenario nodes from other nodes since they are goals that have mutually exclusive contexts and identify sub-missions that cannot be jointly realized.

— *Refining mission specifications out of a library of goals.* We introduce an algorithm that automatically refines the leaf nodes of a CGG using the goals in a library, so that "abstract" goals in the CGG can be further implemented (refined) by more "concrete" goals.

By formalizing the mission specification with a CGG, CROME also offers the following capabilities:

— *Requirement conflict identification.* By checking the satisfiability of the CGG contracts, we are able to identify the presence of conflicts in the mission requirements and immediately inform the designer, before attempting at synthesizing a controller.

— *Realizability checking and controller generation.* CROME checks the realizability of each scenario in the CGG and informs the designer of which sub-goals can be realized (i.e., a controller can be synthesized), given a model of the environment. For each realizable goal of the CGG, CROME synthesizes a controller in the form of a Mealy machine. The controllers are produced together with the CGG.

We illustrate the effectiveness of our methodology and supporting tool on a case study. Our case study shows that the modularity of the CGG allows efficiently checking the feasibility of a mission. The identification of the scenarios allows analyzing the impact of environment variability on the realizability of the robotic mission. The automatic refinement from a goal library facilitates the reuse of existing goals to implement complex specifications. Finally, mutually exclusive scenarios can point to control architectures that may not have a centralized implementation, while still being realizable in a decentralized fashion.

1.5.6 Dynamically orchestrate controllers for several system specifications providing guarantees on the overall system behavior

Automatically synthesizing robotic mission controllers that are proven to be correct from a formal specification is a hard problem that can be intractable if the mission specification is too complicated. Decomposing a mission specification in various sub-missions can make the synthesis of independent parts of the mission easy to achieve; however, we lose the guarantees on the overall mission correctness.

In paper PAPER F, we address the problem of dynamic switching of controllers of a distributed robotic mission while providing guarantees of correctness on the overall mission specification under certain conditions. Our framework allows us to model the overall mission in terms of mission *scenarios* that are enabled under certain mission *contexts*. We represent the mission in terms of a hierarchy of reusable goals, where each goal is rooted in formal contract-based representations.

The major contribution is the possibility for the robot controller to perform a mission while the context is being changed. Each context is related to one *specification controller*. We have introduced a new formal methodology to automatically switch from one specification controller to another while providing guarantees on the satisfaction of the overall missions.

Overall our framework, named CROME, allows us to: *i) decompose* the mission in several independent clusters, each containing mission scenarios acting under a certain context, *ii) automatically refine* the mission using pre-defined and reusable goals, *iii) synthesize* controllers for each cluster and at different abstraction levels, and *iv) dynamically switch* from one controller to another at run-time while formally proving the correctness of the overall mission.

1.6 Conclusions and Future Work

In this thesis, we have presented the methodologies and techniques that can help us build trustworthy autonomous systems. We have shown how formal methods can be used effectively in any aspect of the design. Starting from system requirements, we have seen how we can produce mission specifications that are sound and complete. When uncertainty arises in what the specification should be, we have used reinforcement learning techniques to let the system learn and adapt in different environmental situations. We have seen how automatically generate correct-by-construction system models from its specification with reactive synthesis. Alternatively, we have seen how model-checking and run-time verification can be used to provide assurances to existing systems. Our contributions are both methodological and theoretical. Our frameworks provide methodologies that can guide the system designer in the engineering of autonomous systems while our algorithms enable the system designer to practically realize them.

Assume-guarantee contracts provide the formal foundations for several of our works, while methodologies such as platform-based design have guided our design process. In the future, we plan to continue leveraging on the formal machinery offered by the algebra of contracts together with core concepts of *modularity* and *step wise refinement*. At the same time, we would like to continue using platform-based design to bridge the gap between specification and architecture while reasoning about abstraction layers, reuse, and compositionality.

Specifically, we would like to investigate further how to combine formal techniques such as reactive synthesis with more flexible ones such as reinforcement learning. On one hand, the synthesis methods provide a strategy, usually a maximal permissive strategy (i.e. the one that restricts the behavior of the environment as little as possible), that is functionally correct, but not necessarily *optimal*. On the other hand, reinforcement learning, given a *reward function*, can find the optimal policy. We plan to combine the two approaches by synthesizing a functionally correct policy and then quantifying its optimality with reinforcement learning.

Chapter 2

Paper A

Formal verification of the on-the-fly vehicle platooning protocol

P. Mallozzi, M. Sciancalepore, and P. Pelliccione

International Workshop on Software Engineering for Resilient Systems. Springer, 2016.

Abstract

Future transportation systems are expected to be Systems of Systems (SoSs) composed of vehicles, pedestrians, roads, signs and other parts of the infrastructure. The boundaries of such systems change frequently and unpredictably and they have to cope with different degrees of uncertainty. At the same time, these systems are expected to function correctly and reliably. This is why designing for resilience is becoming extremely important for these systems.

One example of SoS collaboration is the vehicle platooning, a promising concept that will help us dealing with traffic congestion in the near future. Before deploying such scenarios on real roads, vehicles must be guaranteed to act safely, hence their behaviour must be verified. In this paper, we describe a vehicle platooning protocol focusing especially on dynamic leader negotiation and message propagation. We have represented the vehicles behaviours with timed automata so that we are able to formally verifying the correctness through the use of model checking.

2.1 Introduction

Intelligent and connected vehicles will be key elements of future of transportation systems. Within these systems, vehicles will act as standalone systems and at the same time they will interact each other as well as with pedestrians, roads, signs and other parts of the infrastructure to achieve (even temporarily) some common objectives. Future transportation systems might be then seen as Systems of Systems (SoSs) [87] in which the boundaries will change frequently and unpredictably. Moreover, these systems will need to cope with different degrees of uncertainty both at the level of single constituent systems and the entire SoS. Intelligent transport systems promise to solve issues related to road congestion, environment pollution and accidents for a better and more sustainable future [88]. In order to increase safety, reduce traffic congestion and enhance driving comfort, vehicles will cooperate exchanging information among each other and with the surrounding environment as well.

In this paper, we focus on a specific scenario, namely on-the-fly and opportunistic platooning, i.e. an unplanned platooning composed of cars that temporarily join in an ensemble to share part of their journey. Platooning is one of the promising concepts to help us dealing with traffic jams and at the same time to increase the overall safety while driving. A platoon consists of reducing the distances among following vehicles; it consists of a *leading* vehicle driving manually and one or more *following* vehicles automatically driving and following the leader one after another. This concept has been studied and applied especially in trucks for the transportation of goods [89] with the aim of reducing the impact with air and consume less fuel, but not as much work has been done regarding normal vehicles platooning. Each vehicle must be able to communicate with the others, or at least with the cars adjacent in the platoon. The communication is important because each vehicle needs to adjust the speed and the distance according to the other vehicles information. Also, the leader of the platoon is responsible for managing the overall platoon formation, by accepting new vehicles or responding to vehicles leaving.

Platooning is also a way towards autonomous vehicles since, except for

the leader, the vehicles do not need human intervention during the travel journey. Since human intervention is no longer needed, all decisions must be taken autonomously by the vehicle, and this is a huge challenge for safety assurance. Consequently, on the one side the use of platooning promises to enhance safety, and on the other side safety is exposed to new threats and challenges. It is important to notice that nowadays most of the systems are guaranteed to operate correctly only in certain configurations and within the system boundaries. When these boundaries are removed and the system is exposed to unpredictable and uncontrollable scenarios and environments, safety guarantees no longer hold. This will be one of the greatest challenges of future autonomous and connected vehicles that will cooperate with other vehicles, pedestrians, roads, etc. in a SoS setting.

Although there are different levels of autonomy of vehicles[1], autonomous vehicles can be considered as particular self-adaptive systems [90] since they are capable of adapting themselves at runtime. A connected vehicle beside being self-adaptive is also open to interactions with other vehicles and other elements of the external environment. The unpredictability and uncontrollably of the environment hamper the complete understanding of the system at design time. Often uncertainty is resolved only at runtime when vehicles will face with concrete and specific instantiations of the pre-defined environment parameters. This implies that the certification process for safety has to be extended also to runtime phases.

In this paper, we focus on a platooning scenario where the different vehicle's behaviours are organized in various modes [91]. A mode is a concept for structuring the overall behaviour of the system into a set of different behaviours, each of them activated at different times according to specific circumstances. The behaviour of each mode is then represented in terms of a state machine that captures the behaviour of the system in a specific modality, e.g. during the selection of a leader of the platoon, leaving a platoon, etc. Transitions among states can be triggered by timing constraints or external events. A special transition can lead the system to a different mode: in this case the two states involved are *border states* of the modes. Figure 2.1 shows a vehicle platooning scenario that involves different heterogeneous vehicles. Each vehicle is in a certain mode according to its behaviour; we will describe the modes in more detail later. The communication among the vehicles is represented with dotted blue lines.

In this paper, we formally verify the on-the-fly vehicle platooning protocol through the use of the Uppaal [92] model checker. More precisely we verify the absence of deadlocks in the mode-switching protocol as well as other interesting properties.

The rest of the paper is structured as follows: Section 2.2 presents all the modes of our platooning scenario, Section 2.3 describes some parts of the Uppaal model, and Section 2.4 describes the properties we checked on our model. In Section 2.5 we show the results of a concrete simulation of our model in Uppal. Section 2.6 presents the results of the validation we performed

[1]The National Highway Traffic Safety Administration (NHTSA) has proposed a formal classification system based on five levels: "*U.S. Department of Transportation Releases Policy on Automated Vehicle Development. National Highway Traffic Safety Administration, 2013*".

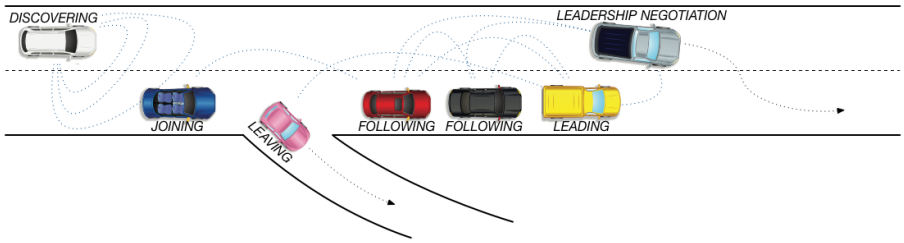


Figure 2.1: Dynamic vehicle platooning scenarios. Each vehicle is in a certain mode according to its behaviour in the platoon.

through the use of the model checker Uppaal. Section 2.7 discusses works that are related to our work and finally Section 2.8 concludes the paper with directions for our future work.

2.2 Multi-Mode System

Partitioning a system into multiple modes, each of which describing a specific behaviour of the system, is a common approach in system design. It leads to a series of advantages, such as reducing the software complexity and easing the addition of new features [91]. A self-adaptive system can be considered as a multi-mode system; if something happens in the environment, the system switches mode in order to adapt to the new conditions. This is the design strategy we follow in this paper.

We start by partitioning our system into different operational modes, recognizing different system behaviours. We have defined the different modes as a set of connected states with common behaviours. There are particular states that we call *border states*: to pass from one mode to another, the system passes through these states. All the modes have one or more *border states* that allow the mode switching of the system. Switching from one mode to another means that the system is passing from one border state of the current mode to a border state of another mode. For each vehicle taking part in the platoon we have identified the following modes:

— *Discovering*: this is the entering mode of the vehicle that wants to take part in a platoon and searches for other vehicles that have the same goals (e.g common destination).

— *Forming*: the first two vehicles that want to form a new platoon enter into this mode. To do that, they decide who will be the leading vehicle of the platoon.

— *Joining*: a vehicle has found an existing platoon and it wants to join it. The vehicle can be accepted in the platoon within a certain time interval;

— *Leading*: the vehicle with the best safety attributes is elected as leader of the platoon. We have assumed that each vehicle shares its safety attributes with the other vehicles. Once in this mode, the vehicle has to steer the following vehicles, propagate information, keep track of the

list of the followers, accept new vehicles that want to join, and, finally, manage the leaving of the followers.

— *Following*: all the vehicles drive in automated manner and follow the leader. A follower can receive information from the leader and propagate it to the other members of the platoon. It also supports the changing of the leader and if the leader leaves then the vehicle goes into the discovering mode again.

— *Leaving*: all the vehicles can leave the platoon at arbitrary time. When the leader leaves, the platoon dissolves. When a follower leaves, it must advise the leader and receive acknowledgement.

— *Dissolving*: vehicle goes in the dissolving mode when (i) it is a follower and does not have a leader anymore or (ii) it is a leader and does not have followers anymore. From this mode, it can either leave or go back to the discovering mode and start a new platoon.

— *Negotiation*: when a new vehicle wants to take part of an existing platoon, either it becomes a follower or it has to negotiate the leadership with the current leader. The vehicle with the highest safety attributes will always be the leader. Leadership negotiation can also be triggered by two platoons that want to merge.

2.3 Uppaal Model Description

Our strategy to model the behaviour of the on-the-fly platooning is to build a generic Uppaal template that incorporates all the modes. This template can be then instantiated for each vehicle that will take part to a specific scenario. More precisely, this model can be instantiated by all the vehicles regardless of their role in the platoon. We can then simulate a variety of scenarios by tuning the vehicles intrinsic properties. This solution is more scalable than having multiple models for different roles of the platoon (leader, follower).

The dynamic leader negotiation is a property of our scenario since we do not know who is going to be the leader beforehand. Furthermore, the leader can be changed during the platoon life. In order to this, we assume that each vehicle has associated a parameter representing its safety characteristics, called safety index, before it enters the platoon. Our models and protocol assure that the leader is always the vehicle with the highest safety index. The safety index it is just a value and it represents the overall safety score of the vehicle, the higher the better. We can assume that this value is calculated taking into consideration all safety-related parameters of the vehicle, either static ones such as the year of the vehicle, the size or dynamic ones taking into consideration the driver experience and the people on board.

In our model every vehicle starts from a *discovering* mode where it looks for other vehicles or platoons to join. In fact, the *formation* of a platoon can happen in different ways:

— Two vehicles negotiating with each other and forming one platoon with one leader and one follower. The two vehicles negotiate the leadership according to their safety index.

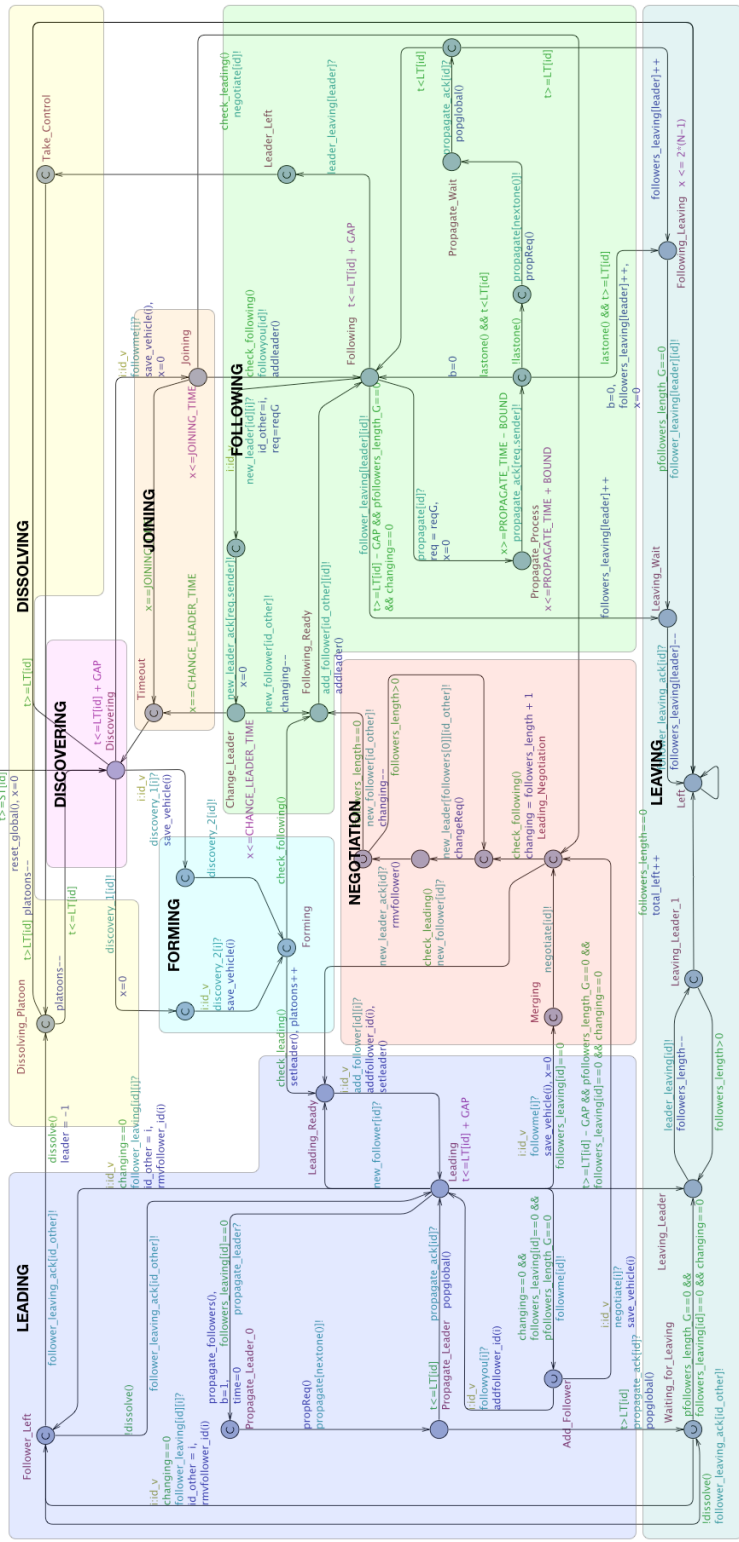


Figure 2.2: Uppaal model with modes.

— One vehicle joining an existing platoon if there is already a formed platoon and the new vehicle is in discovery mode.

— Two existing platoons merging into one after the two leaders have performed a re-negotiation of their leadership.

If the joining of a platoon takes more than the pre-defined constant time (`JOINING_TIME`) to a vehicle, then it goes into discovering mode again. After the formation phase a vehicle can be either in *Leaving* or in *Following* state. The leader keeps track of all its followers at any time by listening to new joining or leaving requests. It can also send messages to all its followers. Message propagation can happen in two ways:

— The leader can reach all its followers and communicate with them all.

— The leader sends a message to the follower immediately behind him and then the message will propagate from follower to follower until reaching the last vehicle in the platoon.

We also take into consideration the propagation time that is needed for a vehicle to pass on the message to the next vehicle. The time is, in fact, crucial for safety-related messages; we want to be sure that the message reaches the whole platoon in the shortest time. We guarantee this by formulating and verifying time-related properties on the message propagation as described in the section below. Another feature of our model is the dynamic leader negotiation also after the platoon has been formed. This can happen in two cases:

— Two platoons want to merge. The platoon with the leader having the highest safety index will take the leadership while the other leader activates the joining procedure to the new leader that has to be completed in `CHANGE_LEADER_ TIME` and afterward it becomes a follower of the newly elected leader.

— A vehicle wants to join an existing platoon and it has a safety index higher than the platoon leader. The current leader passes its followers to the new leader and itself becomes a follower.

2.4 Requirement specifications verified with model checking

The main purpose of a model-checker is to verify the model with respect to a requirement specification. With the timed-automata representation of the system, it is possible to verify safety and behavioural properties of our model such as the absence of deadlocks or the propagation of a safety-critical message within a certain time. Like the model, the requirement specification (or properties) to be checked must be expressed in a formally well-defined and machine readable language. Uppaal utilizes a subset of TCTL (timed computation tree logic) [93, 94]. The path formulae $A <> \varphi$ (or equivalently $A <> \varphi = \neg E[\ ]\neg\varphi$) expresses that φ will be eventually satisfied or more precisely that in each path will exist a state that satisfies φ. The path formulae $A[\ ]\varphi$ expresses that φ should be true in all reachable states.

In order to verify the safety requirements, we have to build a scenario first, i.e., a particular instantiation of the system. Our model is made in order to be configured according to the scenario we want to verify. We first need to set the *number of vehicles* involved and for each vehicle we need to configure few parameters such as its *arrival time*, *leaving time*, and *safety index*. We have automated the configuration process by assigning random values to these values as we explain in the following section. The automation process involves also the properties that are tuned according to the scenario we want to verify. Once we have configured our scenario we can formally verify the following properties:

— *Property 1: If a vehicle is in the leading mode then its safety index is higher then all other vehicles involved in the platoon.*
Assuming a scenario where `Vehicle 3` has the highest safety index the instantiated property would be expressed as:

$$\texttt{A[] (Vehicle(3).Leading} \implies \forall\texttt{(i:id\_v) S[3]>=S[i])}$$

— *Property 2: The propagation of a message from the leader to the last follower happens in a bounded amount of time.*
The time in which the propagation has to happen varies according to the size of the platoon and the maximum acceptable delay is kept by the predefined variable `MAX_PROP_DELAY`.

$$\texttt{A[](b==1} \implies \texttt{time<=MAX\_PROP\_DELAY)}$$

A boolean variable b and a clock variable *time* are two global variables that are used to measure the propagation time from when a message is fired. In order to measure that, when a message starts propagating, the variable `b` is set to 1 while `time` is reset. The properties assures that `time` will always be inferior to the constant `MAX_PROP_DELAY` while `b` is kept to 1. The variable `b` will be reset when the message has reached the last follower of the platoon.

— *Property 3: For each vehicle in the following state exists at least one vehicle in leading mode.*

$$\texttt{A[]}(\forall\texttt{(k:id\_v) Vehicle(k).Following} \implies$$
$$\exists\ \texttt{(i:id\_v) Vehicle(i).}\mathit{All\_Leading\_States})$$

Since the leading mode is formed by a series of states this property is verified by including all the states of the leading mode (as a series of `or` elements). We did not write the full property for readability purposes.

— *Property 4: Whenever the vehicle with the highest safety index starts participating in the platooning it will eventually become the leader.*
Assuming that Vehicle 1 is the one with the highest safety index, the property becomes:

$$\texttt{Vehicle(1).Start} \implies \texttt{<> Vehicle(1).Leading}$$

— *Property 5: For all the path, the vehicle with the highest safety index goes into the leading state.*
Assuming the Vehicle 1 is the one with the highest safety index, the property becomes:

$$\texttt{A<> Vehicle(1).Leading}$$

— *Property 6: All vehicles will eventually leave the platoon.*
Since all the vehicles have a leaving time we can verify that:

$$\texttt{A<> } (\forall\texttt{(i:id\_v) Vehicle(i).Start} \implies \forall\texttt{(k:id\_v)Vehicle(k).Left})$$

— *Property 7: If a leader leaves the platoon then all its followers leave as well.*

$$\texttt{A[]}((\exists\texttt{(i:id\_v) Vehicle(i).Leaving\_Leader} \wedge \forall\texttt{(k:id\_v) Vehicle(k).Following}) \implies \forall\texttt{(j:id\_v) Vehicle(j).Dissolving\_Platoon})$$

— *Property 8: The model is deadlock free.*
Finally, this property assures that for all possible paths there are no deadlocks in our model:

$$\texttt{A[]} \neg \texttt{deadlock}$$

In Section 2.6 we present the verification times of the properties described above. We have noticed that properties apparently very similar require a very different amount of processing time in order to be verified. For example, both properties 4 and 5 verify the leadership of the vehicle with the highest safety index. Property 5 is always verified in less than 1 second, with the time increasing linearly with the number of vehicles. Property 4, instead, can take up to hundreds of seconds with an exponential increase with respect to the number of vehicles.

2.5 Simulation

Latest versions of Uppaal offer the possibility to perform a concrete simulation of the model. It is a verification tool that enables examination of the dynamic executions of a system. The simulation is based on concrete traces, e.g., one can choose a specific time to fire a transition. The tool helps to see at which time a transition can be fired. We have modeled some transition to fire with a uniform probability distribution. For example, in the propagation of the message, the transition will fire somewhere between `PROPAGATE_TIME-BOUND` and `PROPAGATE_TIME+BOUND` time units. We have used these time constraints to verify time properties based on the worst case scenarios when a message has to be propagated from the leader throughout the entire platoon.

In order to perform a simulation, we have to configure our model specifying parameters such as the number of vehicles, starting times, leaving times, and safety indexes. Each vehicle is an instance of the general vehicle template and by launching the simulation we can see how the vehicles interact with

each other. All instances start from the same state and as the time flows Uppaal randomly selects which edge to fire among the available ones of each state. Some edges have guards and invariant in order to model the time of the transition from one state to another as a uniform probability distribution.

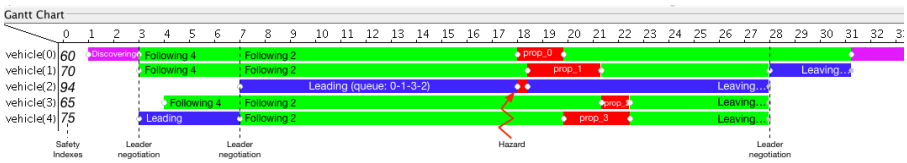


Figure 2.3: Concrete simulation with Gantt Chart in Uppaal.

Figure 2.3 shows the Gantt chart of a simulation. The horizontal axis represents the time span and in the vertical axis the list of vehicles instantiated in the simulation. A vertical line is used to represent the current time (which corresponds to the one displayed in the Simulation Trace-combo box). Horizontal bars of varying lengths and colours represent the different modes of the vehicles. Due to the limited amount of colours we are only able to show a limited amount of modes, specifically: discovering (purple), leading (blue), and following modes (green).

In the simulation showed in Figure 2.3 we can see 5 vehicles participating in the platooning, each with a different safety index. `vehicle0` starts first stays in the discovering mode until other vehicles enter in the platoon. When `vehicle1` and `vehicle4` enter, the three vehicles perform a leader negotiation and `vehicle4` goes starts leading the platoon since it has the highest index. At time 4 `vehicle3` joins the existing platoon until `vehicle2` comes into play and renegotiate the leadership with `vehicle4` and so on. It is also interesting to see the message propagation of a hazard from the leader to all its following vehicles (marked in red).

2.6 Verification results

The simulation shown in Figure 2.3 refers exclusively to a particular scenario. In this section, we instead report the results of an exhaustive verification that we performed on a number of different scenarios. This is obtained by automating the verification process with an external script that is able to generate different scenarios by changing the *number of vehicles* involved in the platoon and by randomly selecting independent variables within each vehicle, such as:

— *Arrival time*: the arrival time of a vehicle;

— *Leaving time*: the leaving time of a vehicle;

— *Safety index*: the safety index of a vehicle.

We are then able to verify all the properties described in Section 2.4 with a number of vehicles from 2 to 5 and for each vehicle configuration we run 100 tests with random scenarios. The height properties are verified by each generated configuration.

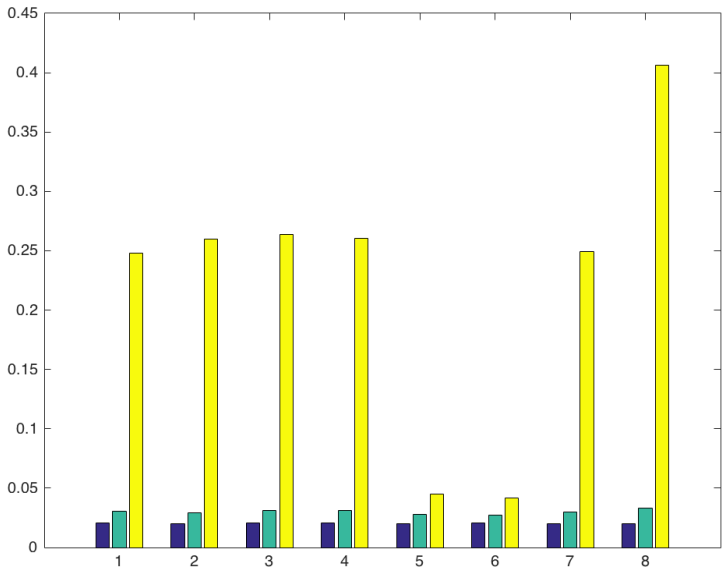


Figure 2.4: Average verification times for 100 iterations. X-axes represent the property being verified. Y-axes the time to verifying it (in seconds). 2-3-4 vehicles scenario respectively

The script generates different models of the system based on a progressive number of the vehicles N and random values of some attributes. It executes two big loops, one to change the random values and one to increment the number of vehicles N. Thanks to the standalone Uppaal verifier, the script verifies the above-mentioned properties with random attribute values of all the models generated. If one property is not satisfied, the standalone verifier generates the counterexample, which is useful to understand why the property is not satisfied. Counterexample files can be open within the GUI of Uppaal. In the end, the script generates a report of the verification, i.e., a text file that traces all the properties, both if they are satisfied or not.

Figure 2.4 reports the time required to verify the 8 properties. The time shown in the figure is the average time required in 100 iterations. Since the time to complete the verification is exponential with respect to the number of vehicles the figure shows the time required by configurations of 2, 3, and 4 vehicles for verifying the 8 properties. For readability purpose, the verification time for configurations of 5 vehicles is not shown in the figure and the average times for 100 iterations are shown in Figure 2.5. As we can see from the figure properties 5 and 6 have times comparable with the verifications times of 2, 3 and 4 vehicles. In fact, these two properties scale linearly while the others scale exponentially.

We have seen how changing the number of vehicles affects the verification time although these change a lot also for every configuration taken into consideration. Within the same number of vehicles, we have performed 100 iterations assigning random values to the vehicle attributes. Figure 2.6 shows how the verification time of a single property with a 5 vehicles configuration is affected by the random assignment of the vehicle attributes.

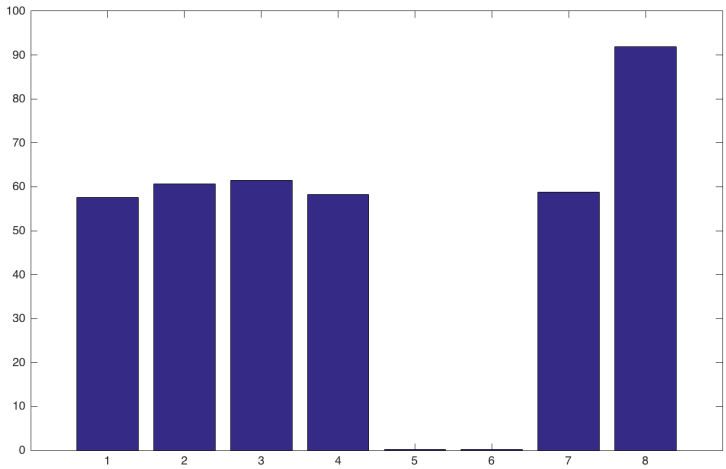


Figure 2.5: Average times of 100 iterations for verifying the properties 5 vehicles.

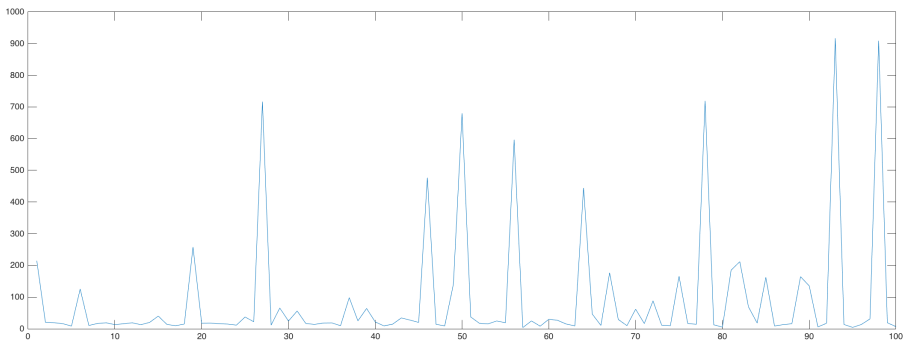


Figure 2.6: Verification times of the deadlock free property for 5 vehicles scenario in 100 iterations.

2.7 Related Works

Kamali et al. [95] have also investigated the verification of vehicle platooning representing it as a multi-agents system. They verified the behaviour partly on the actual agent code and partly with Uppaal with timed-automata abstractions by using two different models, one for the follower and one for the leader.

One of the main challenges in open and self-adaptive systems is to certify that the system is always in a safe state. Since safety cannot be completely evaluated and assured at design time, at least part of the safety assurance must be shifted at run-time. The first ideas for certifying safety at runtime were introduced by Rushby [96], [97]. He proposes an initial idea to certification based on formal analysis at runtime; however much work must be done to produce a solution that can be used concretely.

A promising approach to deal with safety certification at runtime is ConSert [98]. ConSert introduces the idea of *Conditional Safety Certificates* to facilitate the certification of open adaptive systems. Each subsystem is certi-

fied by a modular safety certificate based on a contract-like approach. The evaluation and the composition of the modular certificates happen at runtime. This framework offers flexibility as allows designers to specify safety through variable safety-certificates. Within the approach, all the configurations that a component of the system can assume must be predefined at design time in order to be certified "safe" at runtime. It allows emergent adaptive behaviours only if they can be tamed in certain boundaries with the concept of safety cages. Fully emergent behaviours are not possible to certify with ConSert hence ensuring safety in these cases is a much more difficult problem. A possible research direction can be investigating the theoretical assume-guarantee framework proposed in [99]. This framework allows one to efficiently define under which conditions adaptation can be performed by still preserving desired properties. The framework might provide the infrastructure to automatically calculate at runtime which properties are verified in specific scenarios. For instance, this might suggest excluding some vehicles from the platooning since their inclusion might compromise important properties.

Regarding the automotive domain a more practical approach is the one proposed by Kenneth Östberg and Magnus Bengtsson [100]; they deal with run-time safety by extending the AUTomotive Open System Architecture (AUTOSAR [101]). Claudia Priesterjahnr et al. [102] tackle the runtime safety problem at a component level performing a runtime risk analysis. When a system is trying to connect to another system (for example in a platoon) it computes all reachable configurations and, for each of them, it computes the hazard probabilities at runtime in order to judge whether the configuration is safe or not.

2.8 Conclusion

In this paper, we have presented the formal verification of on-the-fly vehicle platooning. We have modeled the vehicle behaviours with timed-automata so that we were able to verify the correctness of the protocol with model checking. We were able to verify that some properties always hold for a different number of vehicles each with random attributes. All the vehicles are modeled with a unique generic Uppaal model that can be instantiated for each specific vehicle. In this way, it is possible to simulate different scenarios and the verification is easily scalable to more vehicles. Each scenario has been generated with a script, which changes parameters such as the number of vehicles and the attributes for each vehicle and then it verifies that all the proeperties hold. We have focused our attention only to some interesting part of the model such as the dynamic leader negotiation and the message propagation of the vehicles leaving other parts to be further exploited. As future work, we plan to refine our model by releasing some assumptions made during the creation of the model and verifying more properties. As a long term goal, we plan to experiment with the protocol by using a set of miniature vehicles.

2.9 Acknowledgement

This work was partially supported by the NGEA Vinnova project and by the Wallenberg Autonomous Systems Program (WASP).

Chapter 3

Paper B

MoVEMo - A structured approach for engineering reward functions

P. Mallozzi, R. Pardo, V. Duplessis, P. Pelliccione, and G. Schneider

International Conference on Robotic Computing. IEEE, 2018.

Abstract

Reinforcement learning (RL) is a machine learning technique that has been increasingly used in robotic systems. In reinforcement learning, instead of manually pre-program what action to take at each step, we convey the goal a software agent in terms of reward functions. The agent tries different actions in order to maximize a numerical value, i.e. the reward. A misspecified reward function can cause problems such as reward hacking, where the agent finds out ways that maximize the reward without achieving the intended goal.

As RL agents become more general and autonomous, the design of reward functions that elicit the desired behaviour in the agent becomes more important and cumbersome. In this paper, we present a technique to formally express reward functions in a structured way; this stimulates a proper reward function design and as well enables the formal verification of it. We start by defining the reward function using state machines. In this way, we can statically check that the reward function satisfies certain properties, e.g., high-level requirements of the function to learn. Later we automatically generate a runtime monitor—which runs in parallel with the learning agent—that provides the rewards according to the definition of the state machine and based on the behaviour of the agent.

We use the Uppaal model checker to design the reward model and verify the TCTL properties that model high-level requirements of the reward function and Larva to monitor and enforce the reward model to the RL agent at runtime.

3.1 Introduction

The behaviour of autonomous robotic systems is traditionally designed manually and their correctness relies on extensive testing of their requirements. Other approaches can generate optimal controllers of the robot from a synthesis process that starts from the specification of the system [103, 104]. However, the presence of uncertainty makes it hard to model all the requirements at design-time (the partial knowledge of the environment and its dynamic nature will make the specification necessarily incomplete).

Uncertainty may come from the system's environment, unavailability of resources, the difficulty of predicting users' behaviour, etc. [105–107]. Consequently, it is extremely difficult to anticipate all the possible and subtle variations of the environment at design-time. In a goal-oriented approach, we set the objective that we want the system to achieve without specifying how. By using *model-free reinforcement learning* [10] we let the agent explore the environment by trial and error, ultimately producing an optimal policy according to the predefined goal. The policy is learned by interacting with the environment and collecting a reinforcement signal, i.e. a numerical *reward* for each action that the agent performs on it.

An incorrect specification of the reward function, and consequently a gap between the designer's intention and the specification, can cause unexpected behaviours in the agent. One of the problems is *reward hacking* [108], meaning that the agent, by taking into account the reward function, manages to get a high reward without achieving the designer's intentions; this is because it might

optimize towards the rewards function that is indeed not exactly representing the designer's intentions. For example, in a cleaning robot setting, if the reward function gives a positive reward for not seeing any dirt then the agent might learn to disable its vision rather than cleaning up. Instead, if the reward is given only when the robot actually cleans up then the robot might learn to make a mess first and then cleaning up so that it keeps receiving more and more reward.

By embedding more domain knowledge in the reward function one could avoid the reward hacking phenomenon; this would also help the agent to learn the desired policy faster [85]. However, as reward functions become more complex, in turn, it becomes harder to spot mistakes and to be confident whether the reward values that are finally sent to the agent actually reflect the designer's intentions.

Our work goes in the direction of a better structuring of the reward function with the aim of closing the gap between the designer informal goals and the reward signal. Our contribution is the design and validation of a software infrastructure that enables the verification and enforcement of reward functions to an RL agent. From a high-level perspective our approach, which we have called MoVEMo, consists of four steps:

1. *Modelling* complex reward functions as a network of state machines.

2. *Formally verifying* the correctness of the reward model.

3. *Enforcing* the reward model to the agent at runtime using a monitoring and enforcing approach called Larva.

4. *Monitoring* the behaviour of the agent as it transverses the state machines to collect the rewards.

Steps 1 and 2 are performed by the designer that iterates the reward function model until it is compliant with the high-level properties that she/he expresses. Steps 3 and 4 are automatically derived and performed from the reward model.

We have validated our approach in an autonomous driving scenario using the TORCS [109] simulation environment. After modelling the goal of the system with our approach, the reinforcement learning agent learns how to steer, accelerate, and break. We are able to detect bugs in reward functions before enforcing them to the agent. We have packaged the software infrastructure in a stand-alone docker image [110].

The rest of the paper is structured as follows. In Section 3.2 we introduce background information needed to understand our approach, such as reinforcement learning, formal verification, and runtime enforcement. In Section 3.3 we give an overview of existing methods to convey rewards to the agent and the problems that come with it. In Section 3.4 we present the four phases of our approach: modelling, formal verification, enforcement, and monitoring. Finally, in Section 3.5 we present a case study in a racing car simulation environment. We conclude and depict future research directions in Section 3.6.

3.2 Background

3.2.1 Reinforcement Learning

Reinforcement learning is a machine learning technique that involves an agent acting in an environment by choosing predefined actions with the goal of maximizing a numerical reward.

At each time step t an agent receives an observation from the environment that it associates with a certain state. It chooses an action from that state and applies it to the environment that moves to a new state. A reward associated with this transition *state-action-newstate* is determined and sent back to the agent.

In model-free RL, a learning agent starts with no prior knowledge about the environment and, as it receives observations, it tries actions and then collects a reward. It selects its actions by exploiting the knowledge from its past experiences but also by exploring the environment by trial and error learning. The environment evaluates the action taken by the agent at each step by sending back a reward signal to the agent. The goal of the agent is to maximize the expected rewards over time, also known as *return*.

The agent does not learn the reward function but instead it learns the *Q-values*, which are numerical values associated to each action that it can take in a state. In Q-learning [81], a model-free reinforcement learning algorithm, at each time step the agent updates the Q-values associated to the state-action pair as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \cdot max_a(Q(s_{t+1}, a)) - Q(s_t, a_t))$$

where:

— r_t is the reward for the current agent's state;

— s_t is the current state of the agent;

— a_t is the action picked by the agent;

— α is the learning rate; it indicates how much the agent will consider the newly acquired information into its previous state-action value;

— γ is the discount factor; If the factor is close to 0 the agent will only consider the current reward; contrariwise, if the factor is close to 1 the agent will try to maximize the long-term sum of the future rewards.

In this paper, we use the Deep Deterministic Policy Gradient (DDPG) algorithm, which uses the ideas of Q-learning in a continuous action domain [111]. It is an *actor-critic* algorithm and it uses two neural networks, one for the actor and one for the critic. The actor is a policy: it produces the action a given a state s. The critic is the value function: it estimates the action-value function $Q(s, a)$. The critic estimates the value of the current policy by Q-learning, so using the rewards from the environment to improve its estimations. The critic also provides a loss function to the actor that updates its policy in a direction that improves the Q value.

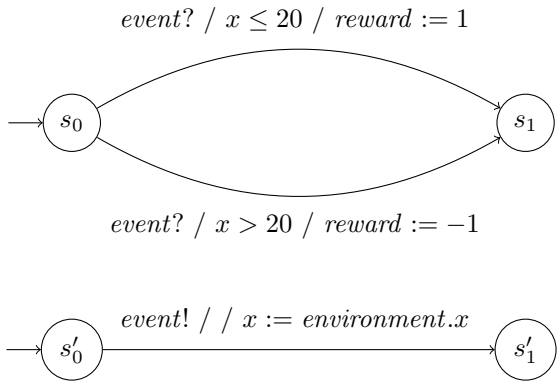


Figure 3.1: Example of a network of UPPAAL automata

3.2.2 Formal Verification

Depending on the system to be analyzed and the kind of properties to be proved, different formalisms are used for their description, giving also rise to different verification techniques. One common technique consists in using finite state machines (e.g., *automata*) to model the system and temporal logic to describe the specification or properties. Formulating specifications in temporal properties is an error-prone task that requires mathematical expertise. In order to facilitate the task of specifying formulas in a correct and accurate way, we plan to use user-friendly notations and approaches, like [112, 113].

In this paper, we use the model-checker UPPAAL [114] to guarantee that the reward function of the reinforcement learning algorithm complies with certain requirements. In UPPAAL the system is modelled as a network of *timed automata* [115]. These automata are state machines where nodes can be labelled with invariants and transitions are labelled with synchronization primitives, guards, and updates. It also allows the definition of discrete variables, and in particular, a special kind of continuous variables called *clocks* to measure the pass of time. Synchronization is carried out using channels. There are two synchronization primitives associated with channels. For a channel c, we can send information to the channel ($c!$) or receive from it ($c?$). Receiving is a blocking primitive, therefore the automaton waits until another automaton sends information to the channel. Guards are Boolean expressions involving any of the variables of the automaton. Likewise, using updates we can modify the value of variables of the automaton. Figure 3.1 shows an example of a network of two automata, the labels on the transitions are in the form of *synch/guard/update*.

In this network of automata, there is a channel *event*, and two integer variables x and *reward*. Intuitively, the top automaton models a reward function, and the bottom automaton the environment in which the reinforcement learning agent is working. Initially, the automaton at the top is waiting for an event to occur, since it is in state s_0 and both transitions are labelled with *event*?. When the bottom automaton—initially in state s'_0—changes state, it sends a message to the channel event (*event*!) and updates the value of variable x of the automaton with the value of variable x in the environment ($x = environment.x$).

The automaton modelling the reward function now reads the value of x and depending on whether it is greater than 20, it gives the agent a positive or a negative reward.

In UPPAAL, properties to verify are written in Timed Computational Tree Logic (TCTL) [116]. This logic consists of the usual propositional logical connectives: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and some operators to quantify over the execution paths of the system and their states. In this paper, we only make use of the operator AG—Always Globally, or, more formally, for all execution paths and in all states. The meaning of this operator is better explained with an example. Consider the automata above, the following TCTL formula checks that *for all executions paths and all states in the system the value of reward will always be either -1 and 1*,

$$\mathsf{AG}(reward = 1 \vee reward = -1)$$

this formula will be satisfied in our model of the reward function. We can also specify properties of states where some condition holds. Consider the following property *if x goes above 20 the agent should always get a negative reward.* This property can be expressed as follows:

$$\mathsf{AG}(x > 20 \Rightarrow reward < 0)$$

This property only considers the state of the execution paths where $x > 20$. Again, it is easy to see that the previous property is satisfied in the system.

UPPAAL offers the possibility of defining functions that can be associated with the *guard* or *update* fields of the automata. This is very convenient as the user can specify complex reward functions in a syntax similar to C and then use the model checker to verify that the specified properties hold across all possible input values.

As we show in the following sections, we formalize the high-level requirements of the reward function in TCTL so that we can automatically check that our model of the reward function satisfies them. Nevertheless, there is still a gap between the model of the reward function and its implementation. We bridge this gap using runtime enforcement tools.

3.2.3 Runtime Enforcement

Runtime enforcement is a technique that ensures that the behaviour of a system—to which we do not have access a priori, i.e., we can only access the system after it has been deployed and is running—complies with certain properties. Runtime monitors are small programs that passively run in parallel with the system while the enforcers act on the monitored systems. Monitors observe the system and based on their perceptions, influence the behaviour of the system to guarantee that some properties are satisfied.

The tool LARVA [117] offers the possibility of defining runtime monitors using special kind of automata called *DATEs* (*Dynamic Automata with Timers and Events*) [118]. DATEs can be automatically compiled into an executable Java program. All elements in a timed automaton can be converted into a DATE (see Section 3.4.3 for the details), which makes LARVA an ideal candidate to define (and generate) the monitor for reward function specified using timed automata. DATEs consist of states and transitions labelled with triples *event/guard/update*.

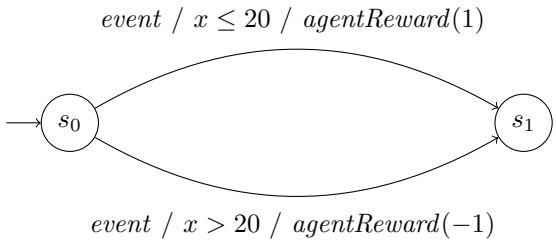


Figure 3.2: Example LARVA automaton

The labels on the transitions mean that if a matching event occurs in the system and the guard—based on event parameters and the automaton state—holds, then the update is carried out and the current state of the automaton updated.

Consider the LARVA automaton[1] in Fig. 3.2. This automaton is almost identical to the reward function UPPAAL automaton presented above. There are only two differences: i) *event* represents that the transition that will be triggered every time the monitor observes that the reinforcement learning agent performed an event; ii) the function *agentReward* sends the reward to the reinforcement learning agent, instead of storing it in a local variable. As in UPPAAL automata, transitions are triggered if the guards are true.

We translate UPPAAL automata to LARVA (we describe the details of the translation in Section 3.4.3) to automatically generate a Java monitor. This monitor will run in parallel with the learning agent. LARVA automatically instruments the monitor and the learning agent so that the monitor can observe the events that the agent performs and its impact on the environment. Based on the events and the observation the monitor will provide the agent with the corresponding reward. The reward that the monitor produce is guaranteed to comply with the high-level requirements of the agent since the monitor behaves as defined in the UPPAAL automaton which has been formally verified.

3.3 Reward engineering: state of the art

3.3.1 Conveying rewards to the agents

Daniel Dewey stated the *reward engineering principle* [119] as follows: as reinforcement-learning-based AI systems become more general and autonomous, the design of reward mechanisms that elicit the desired behaviour becomes both more important and more difficult.

In reinforcement learning the only way to convey the goal that the agent has to achieve is the reward signal, which determines the success of an action's outcome. The system designer has to engineer the reward function so that it encodes the informal objective that he wants the agent to achieve. He has to translate these goals into a numerical form, rewarding the agent for acting *good* and penalizing it for acting *bad*.

[1] In what follows we will use both "DATEs" and "LARVA automata" to refer to the underlying specification language of LARVA.

A well-defined reward function has demonstrated to be successful in several cases such as Atari games [120] and board games [120]. These examples show that a simple reward function, such as the score of the game, can teach the agent to achieve the optimal policy. However, in many tasks the *right* reward function is less clear. In more complex domains such as in automotive, we need more complex functions to produce the desired behaviour of the agent. For example, let us assume that we want to make an agent steering a car. Then, imagine that as reward function we only model the travel time from a point A to a point B. The agent might take an almost infinite amount of trials to drive properly, since it will have to try different combinations of steering, accelerating, and breaking actions.

When the action domain of the agent is continuous, a common way to convey the right rewards is to elicit it from demonstrations of an expert. In methods such as apprenticeship learning, the reward function is learned from observations [121]. The idea is that we take as given an expert optimal policy and we determine the underlying reward structure. This problem is also known as *inverse reinforcement learning*. The given policy follows some optimal reward function but there is no need to articulate it. The agent will derive it by seeing demonstrations.

Another issue with determining the right reward is that the agents need an early feedback on the success of their actions without having to wait for the end of the task. Reward shaping has been addressing such challenge by providing guidance to the agent and incorporating prior knowledge in the reinforcement learning. For example, in potential-based reward shaping we provide heuristic knowledge by an additional reward $F(s, s') = \gamma\phi(s') - \phi(s)$ when moving from state s to s'. Where $\phi(s)$ is a potential function associated with the state s. Prior knowledge can also be encoded directly into the initial Q-values of the agent, which can be equivalent to shape the reward by using a potential function [122].

Multiple sources of rewards can also make the reward function more robust and difficult to hack, as also proposed by Amodei et al. [123]. The reward signals might be independent, complementary, or conflicting with each other. When the RL agent has to deal with multiple reward signals we refer to Multi-Objective Reinforcement Learning (MORL) [124]. There are two main categories in MORL depending on the number of policies to be learned by the agent: single-policy and multiple-policy approach [125]. In our approach, the RL agent learns a single policy based on multiple sources of rewards.

3.3.2 Unexpected behaviours

The encoding of system goals into a reward function can lead to unexpected behaviours in the agent, either because the designer does not include or have the correct information or because she/he makes mistakes during the design of the reward function. Amodei et al. [123] point out some of the major problems in achieving safe and expected behaviours in machine learning agents, such as avoiding *negative side effects* and avoiding *reward hacking*.

The first problem emerges in large environments when the designer of the reward function focuses on few aspects of the environment omitting others. The agent might manage to achieve the designer's goal by doing something

unrelated or destructive to parts of the environment that the designer did not include in the reward function. Basically, the designer should include constraints to what the agent can and cannot do in the environment and not only inferring the system goal. However, in large environments, it might not be possible to identify all the constraints.

We have talked about how simple reward functions such as the score of the game can lead the agent to play several ATARI games. This is not always the case as the agent might simply learn to maximize the reward function without satisfying the specifications of the game (*reward hacking*). For example, in CoastRunner, a boat race game, a human player understands that the goal is to finish the race as quickly as possible while collecting points on the way. The reinforcement learning agent instead keeps hitting some targets on the way without finishing the race because it learns that by doing so it can gain a higher score[2]. There is a wrong assumption here: the reward function does not properly reflect the informal goal of the game to finish the race but rather to simply maximise the score.

Finally, we have a designer that wants the agent to accomplish a certain objective. He has to encode the objective as a reward function and convey it to the agent. We have seen how this encoding can lead to unexpected behaviour in the agent, either because the designer does not include or have the correct information or because he makes mistakes in the designing of the reward function.

3.4 MoVEMo

We propose an approach called MoVEMo to *model*, *verify*, *enforce* and *monitor* reward functions. MoVEMo goes in the direction of carefully engineering complex reward functions by using formal methods combined with the use of multiple reward signals. Our approach is shown in Figure 3.3 and consists of four main steps:

1. The designer graphically models different goals that want the system to achieve as state machines using Uppaal, hence creating a *reward model*.

2. The designer expresses the requirements of its reward function in terms of TCTL properties. Uppaal automatically verifies that the properties hold in the *reward model* across all possible inputs that the environment could issue.

3. The *reward model* is automatically translated into a Larva model preserving the original behaviour. This model interacts with the RL agent at runtime issuing rewards.

4. The Larva model also serves the purpose of *monitoring* the behaviour of the RL agent.

[2]Faulty reward functions `https://blog.openai.com/faulty-reward-functions/`

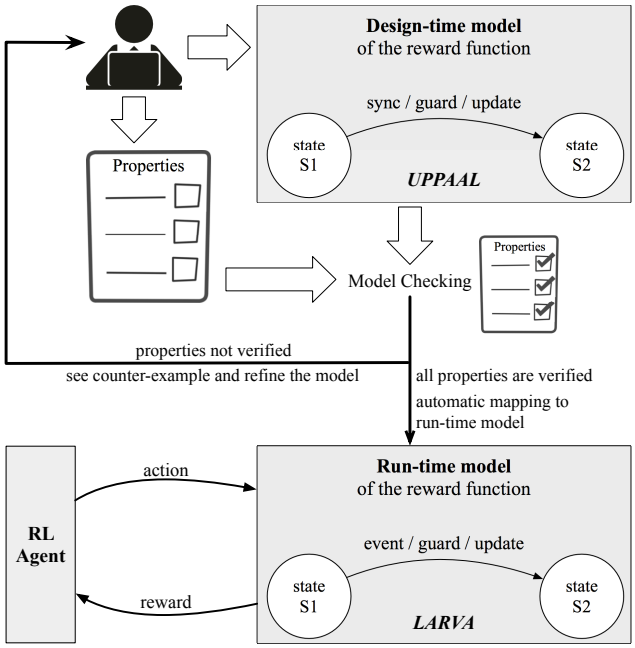


Figure 3.3: Proposed approach: MoVEMo

3.4.1 Step 1: From requirements to reward function

Traditional software development cycle starts with the definition of high-level requirements of the system that are then broken down into smaller objectives that have to be achieved by the individual components. Goal-oriented approaches can be helpful in refining high-level functional requirements into operational goals and non-functional requirements in system invariants [24, 126, 127].

In this phase, after an initial elicitation of the high-level goals to be conveyed to the agent, the designer models them as automata in Uppaal. Each goal is represented by a separate automaton whose states encode a configuration of the agent in the environment. A reward function is associated with each state.

At all times, the state of the agent resolves in multiple states in the monitor, one state for each automaton. Each state is issuing a reward proportional to how distant is the agent to the goal modelled in the automata. We compute a single scalar value from all individual rewards that we then feed to the agent as in the standard reinforcement learning framework.

3.4.2 Step 2: Verifying the requirement

The designer expresses the reward function requirements in terms of TCTL properties. Furthermore, all the environment variables that interact with the reward model have to be expressed as simple automata that can produce values within a certain range. Uppaal can then formally verify the compliance of the *reward model* with the properties expressed by the designer.

In case any of the properties is not verified, Uppaal will show a coun-

terexample. The designer is then able to understand in which case its reward function does not hold the specified property. This is a very useful feature that allows the designer to go back and iterate on the reward model until she/he is satisfied with the result.

3.4.3 Step3: Enforcing the reward function

To enforce the reward function we convert the UPPAAL model into a LARVA monitor. To do so, we have established mapping rules between UPPAAL and LARVA automata. More specifically:

— *Channels* in UPPAAL automata become event listeners in LARVA. In UPPAAL the reward model *sync* with the environment model. In LARVA the *events* are automatically fired from the real or simulated environment values.

— *Guards* on the transitions of UPPAAL automata are directly mapped as *guards* in LARVA automata.

— *Updates* in UPPAAL are also mapped directly to updates in LARVA, which can be associated with actions such as the execution of an arbitrary program. This transformation also requires defining the same variables in both automata. For example, the variable associated with the rewards in UPPAAL are converted into equivalent variables in larva that are then updated and sent to the RL agent as part of a LARVA update.

— *Clocks* in UPPAAL become timers in LARVA.

Following the previous steps, the UPPAAL model is automatically translated into a LARVA DATE that issues rewards to the RL agent at runtime.

The LARVA model encapsulates all the states in which the agent can be at any time in the environment. This model can also be used to monitor the agent behaviour. At the moment the model only collects data on which state the agent visits and issues a reward for each state. However, having a model that encapsulates the possible behaviours of the system into states can serve the purpose of preventing the agent to reach *bad* states. We are currently exploiting such aspects of *preventive monitoring*.

3.5 Autonomous Driving with TORCS

In order to validate our approach, we have used The Open Racing Car Simulator (TORCS) [109] to simulate an environment where a RL agent can apply its action and learn how to drive. We have used the DDPG algorithm [111] as the decision-making policy of the agent. We have extended an existing implementation [128] of the DDPG in order to support external rewards coming from the LARVA model. We have encapsulated all the software platform in a docker image so that it is very easy to build a reward function, lunch a simulation and collect the results [110].

The RL agent uses the following signals from the simulation environment:

— *TrackPos*: Distance between the car and the track axis.

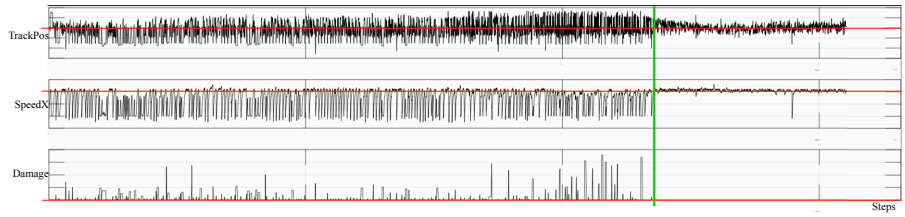


Figure 3.4: Some of the monitored values from one of the simulations in TORCS. The red lines are the goals inferred by the UPPAAL reward model.

— *Track*: Vector of 19 range finder sensors around the vehicle. Each sensor returns the distance between the track edge and the car within a range of 200 meters.

— *Opponents*: Vector of 36 sensors around the vehicle. Each sensor returns the distance of the closest opponent within 200 meters range.

— *Damage*: Current damage of the vehicle, the higher is the value the higher is the damage.

— *Angle*: Angle between the car direction and the track axis

— *SpeedX*: Speed of the car along its longitudinal axis.

Based on the above observations the RL agent can apply the following actions:

— *Steering*: The steering value can be between -1 (full left) and 1 (full right).

— *Accelerating*: The value of the virtual gas pedal can be between 0 (no gas) and 1 (full gas).

— *Breaking*: The value of the virtual break pedal can be between 0 (no break) and 1 (full break).

3.5.1 Conveying the goals to the agent

Each state provides a reward value that should reflect how much the agent is compliant with the goal associated with the automata that the state belongs to. The reward value can be a simple scalar or it can come from a complex function. Below we describe the goals we have modelled.

Staying in the middle of the lane

This goal corresponds to try to keep the value of $TrackPos$ equal to zero. We have modelled 4 states in which the vehicle can be: `CenterRoad`, `LimitRoad`, `RightOffRoad`, `LeftOffRoad` according to the position of the vehicle on the road. In each state the reward function is proportional to the error with $TrackPos$. Each state has additional rewards according to how far the car is from the centre. Furthermore, we take into consideration the previous action

taken by the agent, penalizing it more if it keeps steering towards the wrong direction.

Keeping a certain speed

We want the agent to keep a constant speed of 100 Km/h, slowing down when approaching a curve. For this goal we have modelled 3 states in UPPAAL, according to the state of the vehicle in the road: `GoingStraight` and `Curve`. The reward, when there are no curves ahead of the vehicle, is proportional to the error related to the goal of keeping the speed at 100Km/h. In order to detect if a curve is in front of the car, we have used the 19 range finder sensors around the vehicle (*Track*) to build a function that is used in the guard to transit to the state `Curve`. Since we want to stimulate the car to slow down when a curve is detected, the reward function of this state is proportional to the error of a lower speed than the goal speed of 100 Km/h.

Avoid damages

We have modeled this goal in two states: `Damage` and `Normal`. The reward function penalizes the agent by a constant value every time it receives a damage, this can happen by hitting other vehicles or going off-track.

Avoid getting stuck

When the LARVA monitor detects that the vehicle keeps going at a very slow speed for a while, it might be because the vehicle is in a state where is trying actions but it is physically blocked by the limit of the road. We have modelled this situation in a `Stuck` state where the agent gets penalized if it stops going forward. At the same time, it is also encouraged to try hard steering manoeuvres that can get it out of this state.

Avoid other vehicles

When racing with other cars we want to avoid collisions with the other vehicles. We have split this goal into four UPPAAL automata with the purpose of detecting the presence of other vehicles in the four sides of the car: *Ahead*, *Behind*, *Left*, *Right*. Each automaton is composed of several states according to how distant are the other cars to the side of the vehicle, a reward function is assigned to each of these states penalizing the agent for being close to other cars.

Furthermore, in the presence of other vehicles, staying in the middle might not always be the best policy. The RL agent might want to overtake or simply avoid collisions with other vehicles by going to the right or to the left of the road. This is why, when other vehicles are racing, we update the *TrackPos* goal according to which side of the road is free to drive. For example, if a car is detected on the right side, we reward the agent for driving to the left side, and so on. This is directly encoded in the UPPAAL model so we do not need to re-run the verification of the properties.

3.5.2 Verifying properties

Combining multiple sources of rewards, each contributing with complex functions, will create a big reward model with many states. As we have chosen UPPAAL as modelling environment, we can verify that the properties hold across all possible states of the system.

We have only modeled the signals that the reward model uses to compute the reward such as `TrakcPos`, `SpeedX`, `Angle`, `Damage`. After discretizing each signal with a step value and a range with a lower and upper bound, we are able to run simulations in UPPAAL and verify that the properties that we specify hold across all possible states. In our example, we have modelled 6 automata with a total of 21 states.

We have verified several high-level design goals that helped us fixing mistakes or bugs in the reward functions that we have encoded in UPPAAL. Though we could have assigned any value as a reward, in our model we have assigned only positive values to states of the system that are *good* and negative values when it deviates from the goal of the system. In the following, we describe two examples of TCTL properties that we have verified in the reward model for TORCS.

If the deviation from the speed goal is more than 10 km/h, the reward associated with the speed should be negative

$$\begin{aligned}&\mathsf{AG}((Speed.goingStraight) \wedge \\ &\quad (goalSpeedStraight - speedX > 10) \\ &\quad \implies speedX_{reward} < 0)\end{aligned}$$

where *Speed* is the automaton connected to the speed goal and it is in the state *GoingStraight*, where there are no curves ahead of the car. *goalSpeedStraight* and *speedX* are two variables representing respectively the target speed for when the road ahead is straight and the current speed of the vehicle.

If the vehicle is proceeding in the centre of the road, it has no damage and with a speed less than 10Km/s from the goal speed, then the overall reward should be positive

$$\begin{aligned}&\mathsf{AG}(TrackPos.centerRoad \ \wedge Speed.goingStraight \wedge \\ &\quad Damage.no \ \wedge (speed_{error} < 10) \wedge \\ &\quad (goalSpeedStraight - speedX > 10) \\ &\quad \implies combined_{reward} > 0)\end{aligned}$$

In this case we take in consideration three goal models $TrackPos$, $Speed$, $Damage$ and we verify that the combination of all their rewards is positive.

3.5.3 Results

We have compared our UPPAAL reward model (`URM`) with the reward function proposed in [128] that has already been shown to be an improvement of the original reward function proposed by Google in [111]. We will refer to this function as `BRF` (benchmark reward function) and it is expressed as follows:

	Without opponents		With opponents	
	BRF	URM	BRF	URM
Time (h)	3.4	2.1	13.1	12.0
Episodes (#)	302.8	146.2	1312.7	925.7
Center (%)	60.9	83.6	47.3	61.4
Stuck (%)	5.3	4.6	29.7	37.2

Table 3.1: Average results of the Uppaal Reward Model (URM) compared with a Benchmark Reward Function (BRF).

$$R_t = V_x * cos(\theta) - V_x sin(\theta) - V_x * |TrackPos|$$

Where V_x is the vector representing the car velocity, and θ is the angle between V_x and the track axis. The above reward function aims to maximize the longitudinal velocity (first term), minimize transverse velocity (second term) while penalizing the agent if it deviates from the center of the road (third term).

The results show that the agent learns much faster by using reward functions produced through our approach. Table 3.1 shows the average results of *100 iterations*. In order to complete one iteration, the agent must learn how to drive on the track and complete *20 laps*. An episode of the algorithm ends when the vehicle is perpendicular to the track axis.

The first two columns are the results when the car is racing without opponents while the last two columns are the results with other vehicle racing together. Table 3.1 shows the values of Time (in hours) and the number of episodes to complete 20 laps. Furthermore, we see the percentage the agent stayed in the state *Center* of the road (the higher the better) and how much it got the state *Stuck*. We see that with the UPPAAL reward function it achieves its goal faster and with less number of episodes than the simple python function. The agent performs quite well when there are no other cars on the track, but not that good when racing with other vehicles. Dealing with opponents is a more complicated problem and we believe the reward function can be further improved.

Figure 3.4 shows some of the monitored values i.e. `TrackPos`, `SpeedX`, `Damage` of one iteration of the agent using the UPPAAL reward model. The red lines indicate the goals set in the UPPAAL model for each value: stay close to the centre of the road, keep a speed of `100 Km/h` and avoid damages. We can see that the agent starts with a random behaviour, receiving a lot of damages and continuously changing speed as it learns the goals by trial and error. The green line indicates the point when the agent has learned the goals and continues going at the desired speed while avoiding damages and staying more or less in the middle of the road.

3.6 Conclusion and future work

We have presented a framework in which one can model reward functions for reinforcement learning agents as state machines. This framework allows us to verify properties of the reward function at design-time. We have also formalised and implemented an automatic mapping between the design-time model of the reward function and its implementation that is actually exploited by the agent at runtime. Our results show that, by building more complex and robust reward functions, the agent can learn faster to achieve its goal.

This work is a first step toward the engineering of the reward function. This can help the designer eliciting the requirements in terms of goals to be achieved by the agent as well as to identify the possible uncertainties due to, e.g. the unpredictability of the environment.

The reward model can also be used to monitor the behaviour of the RL agent at runtime. As future work, we envision to use such monitor as a safety envelope for the agent. By doing preventive monitoring we can detect anomalies in the behaviour of the agent and prevent potentially dangerous actions to be executed on the environment.

Acknowledgment

This work has been partially supported by: the Wallenberg Autonomous Systems Program (WASP), the Swedish funding agency SSF under the grant *Data Driven Secure Business Intelligence*, the Swedish Research Council (*Vetenskapsrådet*) under grant Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*), and the European ICT COST Action IC1402 (*Runtime Verification beyond Monitoring (ARVI)*).

Chapter 4

Paper C

A runtime monitoring framework to enforce invariants on reinforcement learning agents exploring complex environments

P. Mallozzi, E. Castellano, P. Pelliccione, G. Schneider, and K. Tei

International Workshop on Robotics Software Engineering. IEEE/ACM, 2019.

Abstract

Without prior knowledge of the environment, a software agent can learn to achieve a goal using machine learning. Model-free Reinforcement Learning (RL) can be used to make the agent explore the environment and learn to achieve its goal by trial and error. Discovering effective policies to achieve the goal in a complex environment is a major challenge for RL. Furthermore, in safety-critical applications, such as robotics, an unsafe action may cause catastrophic consequences in the agent or in the environment. In this paper, we present an approach that uses runtime monitoring to prevent the reinforcement learning agent to perform "wrong" actions and to exploit prior knowledge to smartly explore the environment. Each monitor is defined by a *property* that we want to enforce to the agent and a *context*. The monitors are orchestrated by a meta-monitor that activates and deactivates them dynamically according to the *context* in which the agent is learning. We have evaluated our approach by training the agent in randomly generated learning environments. Our results show that our approach blocks the agent from performing dangerous and safety-critical actions in all the generated environments. Besides, our approach helps the agent to achieve its goal faster by providing feedback and shaping its reward during learning.

4.1 Introduction

Artificial intelligence is increasingly being used to solve problems in many different domains, such as robotics, where a software agent is trained to act autonomously in an, often, unknown environment. Reinforcement Learning (RL) [80] algorithms can be used to train a software agent: the agent learns a policy that maximizes a final reward by trying different actions on the environment (*trial and error*) and collecting rewards.

During training, at *learning time*, the agent can perform actions that are potentially dangerous to the environment or to itself. At *execution time* we cannot be sure that the agent will always act correctly since it uses probabilistic models to make decisions. By *safe exploration* [129] we refer to the problem of guaranteeing that an agent has to act safely both at learning and execution time. For example, a cleaning robot should learn to clean the dirt without breaking other elements in a room, or without harming the agent itself.

Runtime verification techniques can prevent the agent to perform catastrophic actions. Safety-critical requirements can be encoded in one or more monitors and enforced at learning and execution time when the monitor detects that the agent is about to violate them. On one hand we have to *convey goals* to the RL agent through the reward function, on the other hand we want the agent to respect some important properties that include safety-critical requirements, which we call *invariants*, at all time. The work in [130] goes in the direction of conveying the goals by building more structured reward functions, by modelling and verifying them at design-time. In this paper, we address the problem of preserving the invariants of a RL agent at learning and execution time with an approach called WiseML. In [131] we have presented a preliminary idea of WiseML, a method that combines model-free RL with runtime monitoring and enforcement, without providing a concrete solution.

In this paper, we further develop the idea, provide a concrete solution, and largely validate it.

WiseML is agnostic with respect to the RL algorithms used to train the agent. We refer to WiseML as a *safety envelope*: it wraps the RL agent and prevents the execution of actions that would violate its invariants. The WiseML enhanced RL agent, which we call WiseML agent, analyses every action that the RL agent proposes and those that do not violate the specified invariants are sent to execution.

We express the invariants via the use of *specification patterns*, which based on the work in [49, 132], we use 4 patterns: *absence*, *universally*, *precedence*, and *response*. Once invariants are expressed in terms of patterns, then they can be automatically translated in Linear-time Temporal Logic (LTL) [133] or other logics thanks to the mappings provided in [49, 132]. We implement each invariant the RL agent has to obey with a monitor. We introduce also a concept of *context* that is similar to the concept of *scope* present in the specification patterns proposed by Dwyer et al. [49]. Each pattern has a scope, which defines in which part of the execution the pattern must hold. In WiseML we have a *meta-monitor* that dynamically activates and deactivates the individual monitors according to their context.

Blocking unsafe actions will prevent the agent from damaging the environment or itself. However, the agent can also *learn* what caused the violation in order to improve its policy in the future. The safety envelope includes a *reward shaping* component that influences the rewards received by the agent at runtime. This component penalizes the agent for attempting to violate specified invariants since the RL agent performs its decisions based on probabilistic models we can never be sure that the agent will never perform harmful actions during execution time. For this reason the *safety-envelope* is active also during execution time. However, in the current version, we have used WiseML only at learning time. In the future, we will also investigate and experiment during execution time.

The approach has been evaluated extending the *gym-minigrid* platform [134] with our environments and the WiseML *safety envelope*. We have evaluated the RL agent in *150 randomly generated* environments of different sizes, each executed 10 times with the presence of WiseML and 10 times without. Our results show that WiseML correctly enforces the invariants in all the simulated scenarios. Furthermore, thanks to the reward shaping feature, the agent learns much faster with the presence of WiseML. In fact, our experiments show that WiseML blocked violations `100%` of the times while helping the agent converging up to `55%` faster with respect to the learning performed without WiseML.

Summarizing, the main contributions of this paper are:

— WiseML, a framework that uses runtime monitoring to prevent wrong behaviours of an RL agent and to convey prior knowledge of the environment to the agent while it is exploring. The approach is completely independent of the RL algorithm chosen to train the agent.

— WiseML contributes shaping the rewards by using invariant violation as punishments for the RL agent.

— To facilitate the specification of invariants we enable the users to specify them via the use of specification patterns.

— An evaluation conducted on randomly generated environments; the agent has only *partial-observability* of the underlying state of the environment.

The paper is structured as follows. Section 4.2 gives an overview of the specification patterns, the reinforcement learning algorithm, and the runtime monitor techniques. Section 4.3 survey related works. Section 4.4 presents our approach. Section 4.5 presents the case study performed on a gridworld environment, explains how the evaluation has been conducted and introduces the results analysed from the collected data. Finally in Section 4.7 we discuss our results and future work.

4.2 Background

4.2.1 Specification patterns

Capturing temporal properties in a concise and correct way is a major challenge [46, 135]. Syntactic correctness can easily be ensured through standard language processing techniques. However, guaranteeing that a property matches a software engineer's intuition is much harder.

Several lightweight specification languages have been proposed in the last years [135–137]. A different approach has been proposed by Dwyer et al. [49], which proposed qualitative property specification patterns in the late nineties. They analyzed a set of 555 specifications from at least 35 different sources in order to define a catalogue of eight qualitative specification patterns[1]. These specification patterns are organized in two major groups: *occurrence patterns* and *order patterns*. Occurrence patterns focus on a single event (or state) during system execution (e.g., absence or existence of an event). Order patterns capture relations of multiple events can emerge during system execution (e.g., response or precedence). Specification patterns are automatically translated to temporal logics and query languages, e.g., LTL and CTL [49].

Qualitative specification patterns have been extended to express real-time properties and the result is a catalogue of real-time specification patterns [138]. They have been also extended to express probabilistic quality requirements (e.g., reliability, availability, and performance requirements) and the resulting catalogue is known as probabilistic specification patterns [139]. Finally, the work in [132] presents a unified catalogue that collects the existing specification patterns and combines them together with 40 newly identified or extended patterns.

4.2.2 Reinforcement learning

In *reinforcement learning* [10] a software agent collects *observations* from the environment and performs actions. Each observation represents a *state* of the environment and as the agent moves from one state to another it collects a numerical *reward*. The goal of the agent is to maximize the reward collected along the way. The environment can be formally described as a *Markov Decision Process* (MDP) [140]. An MDP is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$. At each timestep t the agent interacts with the MDP by observing a state $s_t \in \mathcal{S}$ and by choosing

[1] `http://patterns.projects.cis.ksu.edu/`

an action $a_t \in \mathcal{A}$. The environment in response will transition to the next state s_{t+1} with probability $\mathcal{T}(s_t, a_t)$ and give a reward $r_t \sim \mathcal{R}(s_t, a_t)$. The goal of the agent is to maximize the *return* $G = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$, which is the sum of all the *discounted* rewards, where $\gamma \in [0, 1]$ is known as the *discount factor*. The value of a state $V(s)$ represents how good is for the agent to be in the state s. Formally, it defines the expected sum of rewards from state s.

The agent will learn a *policy* or *value function* used to estimate the action to perform given a state of the environment. It does not learn a model of the environment and so it can not explicitly predict the effect of its action. Instead, it needs to gather actual experience by exploring the environment, which can make the exploration process dangerous. As the agent moves to a real-world environment, it has to respect some safety constraints. An action that violates some safety constraint can cause catastrophic consequences in both the agent and the environment.

4.2.3 Runtime verification

Runtime verification (RV) [20, 21] is a technique based on monitoring software executions. It detects violations of properties, occurring while the monitored program is running, eventually providing the possibility of reacting to the incorrect behaviour of the program whenever an error is detected.

Properties verified with RV are specified using any of the following approaches: (i) annotating the source code of the program under scrutiny with *assertions* [141]; (ii) using a high-level specification language [142]; or (iii) using an automaton-based specification language [143–145].

One way to verify properties at runtime is through the use of *monitors*. A monitor is a piece of software that runs in parallel to the program under scrutiny, controlling that the execution of the latter does not violate any of the properties. In addition, monitors may create a log file where they add entries reflecting the verdict obtained when a property is verified. In general, monitors are automatically generated from the annotated/specified properties [22, 146].

We will here consider the possibility of *monitoring* the execution of a program for different purposes. We may distinguish three different "kinds" of monitoring: (i) *proper monitoring*, where the monitor collects data, eventually performs simple side-effect free computations (e.g., calculate an average during a specific amount of time), sending the data to another device or monitor; (ii) *runtime verification* is concerned with verification of one or more properties about the expected behaviour of the system under monitoring; (iii) *runtime enforcement* is performed by monitors that carry the code to be executed in the monitored system, send specific commands to control the system, or enforce a given property (as mentioned above) not allowing the system to act differently from the specification.

Researchers usually talk about RV without distinguishing between the above three meanings. In this paper we will instead use the term "monitor" to refer to any of the above three specific uses, and we will clarify when confusion may arise (e.g., we might talk about an "enforcer" if we want to emphasize that the monitor is indeed enforcing a property).

4.3 Related Work

The literature on safe exploration has highlighted several directions to address the problem [129, 147, 148]. Thomas et al. [149] focus on ensuring safety with a policy improvement algorithm that provides probabilistic guarantees on the agent policy, given that the environment can be modelled as an MDP, or partially observable Markov decision process (POMDP) [10]. Lipton et al. [150] modified the DQN algorithm with the concept of *intrinsic fear* that shapes the rewards of the agent guiding it away from catastrophes. The agent interacts with the environment through an MDP and the intrinsic fear model is learned using the data collected from a finite sample of states.

Human Intervention RL (HIRL) is an approach by Saunder et al. [151] that uses human overseer to avoid catastrophes in model-free reinforcement learning agents. As the *safety-envelope* proposed by our approach WiseML, the human overseer stands between the agent and the environment and it can either let the agent's actions to be applied to the environment or block them. The decisions taken by the human are used to train a module, the *blocker*, via supervised learning. The main difference with our approach is that they used a trained model to block potentially dangerous actions at runtime. We use a *hand-coding* approach to specifying invariants at design-time. This comes with the trade-off of having less flexibility in terms of recognized violations compared with a supervised learning model, but more assurances in term of safety (since monitors are not based on machine learning models, will always block the violation as specified).

The work in [152] expresses the properties that the agent must satisfy in LTL and produces an MDP, which is the product of the original MDP and a Limit Deterministic Büchi Automaton (LDBA) generated from the LTL properties. In this work one needs to know the complete information about the environment which is modelled as an MDP with labelled *safe* and *unsafe* states. The agent explores only the *safe* parts using Q-learning to learn the optimal policy.

Al-Shedivat et al. [153] focuses on intelligent exploration of the RL agent on complex environment. They takes advantage of a hierarchical framework for RL [154] by training a meta-controller on learning the sequence of known subgoals while a low-level controller learn how achieve each subgoal.

Our approach builds on several of these ideas. We use LTL to easily encode prior knowledge into formal rules and reward shaping as a basic technique to help the agent to reach the goal. In real-world applications, since it is impractical or even impossible to precisely and completely model the environment, the agent partially observes the environment through its sensors. So in our work, the agent has partial observability of the environment. The agent is able to understand the underlying state of the environment by collecting multiple observations and integrating them over-time. This process is performed by *Long Short-Term Memory* (LSTM) [155], a particular kind of *Recurrent Neural Networks* [156] used as main deep learning model of the RL algorithm.

4.4 WiseML

WiseML addresses the safe exploration problem of a RL agent, during and

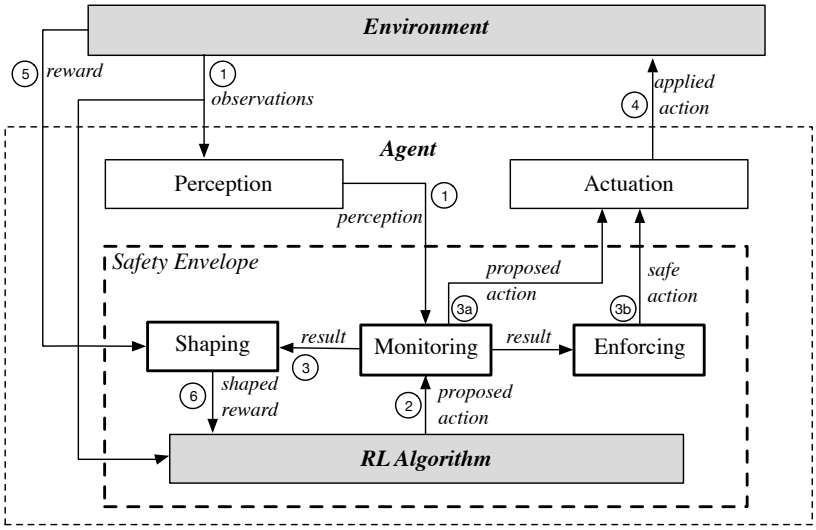


Figure 4.1: Overall architecture of WiseML

after training, in four main directions:

1. *modeling* invariants in terms of property specification patterns [49, 132];
2. *monitoring* the agent in different contexts as it performs actions freely in the environment. We enable the definition of invariants that should be checked and preserved in specific contexts of the environment. This is realized through the use of various monitors that are orchestrated by a meta-monitor.
3. *enforcing* a safe behaviour of the agent when it is about to violate the invariants;
4. *shaping* the reward of the agent so it learns to avoid future bad situations, converging faster to its goal.

Our approach consists of a *safety envelope* around the agent so that it is protected from performing dangerous actions for the environment or itself. Before a violation of any invariant is about to happen, the monitors stop the *unsafe* action from being executed on the environment. The agent still learns from its mistake as WiseML shapes its reward, meaning that the final reward coming from the environment is modified to take into account the blocked violation.

Figure 4.1 shows the main architecture of WiseML. Both the agent and the environment are unaware of the presence of WiseML. In this sense, our approach is agnostic to the RL algorithm used. The *Safety Envelope* surrounds the RL agent. The data between the safety envelope and the environment is processed by the *Perception* and *Actuation* components.

From a high-level perspective, WiseML works as follows. At first, the environment sends the observations of its current state to the RL agent. The perception component analyses raw observations from the environment and

converts them in *perceptions*, more high-level representations of the world outside the agent (*1*). The perceptions are used to model the invariants in the monitors. The monitoring component processes the perceptions and the proposed action by the RL agent. In this phase, the meta-monitor activates the monitors according to their *context* and checks the satisfaction of the monitored invariants. The results of this analysis are sent to the shaping and enforcing component (*3*). Each monitor contributes to computing the overall *shaped reward* (*6*) and the final action sent to the environment (*4*). This action can be either the same proposed by the agent (*3a*) or a *safe action* computed by the enforcing component (*3b*) according to the state of the monitors and their *operational mode*. Each monitor can be configured to be in *shaping* or *enforcing* operational mode. In shaping mode the monitor only influences the reward given back to the agent; in enforcing mode, besides the reward, it also affects the action proposed as explained in Section 4.4.3.

With WISEML we can model each invariant separately as one monitor to be activated in a specific *context*. The meta-monitor will dynamically trigger all the monitors that have the context matching with the current execution of the agent in the environment. A context can be a function of the agent's perceptions, actions or both. It can also indicate that the invariant holds in every situation, regardless of the context. Different monitors can be combined in order to model all the invariants of the agent. For each monitor, the designer can specify the *rewards* to be given to the RL agent in case of violation or compliance with the monitored invariant. WISEML provides a simple interface where the designer can specify all the monitors in a user-friendly `JSON` file. The designer has to specify a few parameters for each monitor in order to activate them as follows: (i) `monitor-name` (ii) `monitor-type`, (iii) `monitor-context`, (iv) `monitor-invariant`, (v) `monitor-operational-mode`, and (vi) `rewards`.

In the following, we will describe in more detail the main components of the safety envelope at the core of WISEML: *monitoring*, *shaping*, and *enforcing*.

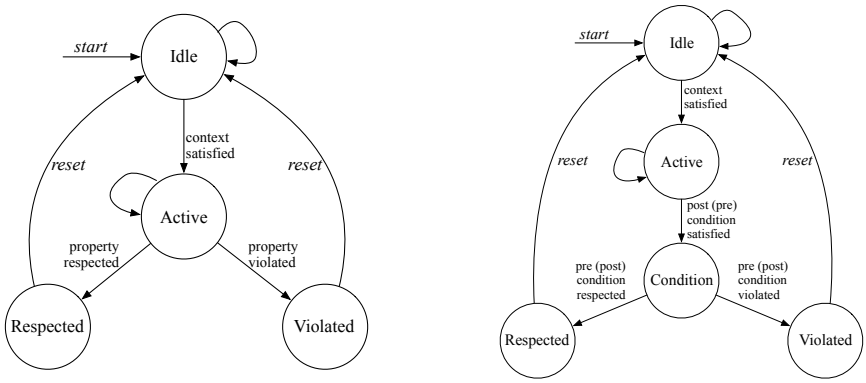
4.4.1 Monitoring

The monitoring component continuously examines the status of the agenxt and of the environment and communicates the results of its analysis to the shaping and enforcing components. The analysis is performed by several monitors, one for each invariant that the agent should respect in a specific context.

Invariants can be expressed as temporal specifications of constraints and preferences, similarly to when specifying properties in Linear Time Logic (LTL) [133]. The objective is to verify that, under specific context, the conditions specified by the system designer are satisfied by the RL agent. A context C is a function $f(s_e)$, while a condition A can be expressed as a function $f(s_e, s_a, a_p)$ where:

- s_e is the current state of the environment (as perceived by the agent through the perception component);
- s_a is the current state of the agent;
- a_p is the action proposed by the agent to be executed on the environment.

The system designer can specify the context and the conditions using one of the available patterns. In the following, we describe the patterns supported by WISEML and the equivalent LTL formula, where A and B are the monitor



(a) Absence/Universally pattern

(b) Precedence/Response pattern

Figure 4.2: Examples of randomly generated environments.

conditions, and C is a context condition as described above. We have chosen the following patterns in order to capture the *occurrences* and the *order* of events and operations that can occur while the agent is training by exploring the environment:

— *Absence:* $C \implies \Box(!A)$.
A is never true. If active, this type of monitor verify that the condition A is false at all times.

— *Universally:* $C \implies \Box(A)$.
A is always true. If active, the universally monitor ensures that the condition A is true at each step taken by the agent.

— *Precedence:* $C \implies \Box(!B\mathcal{W}A)$.
Globally, A precedes B. If active, if B is true then the monitor checks that A has become true in the past.

— *Response:* $C \implies \Box(A \rightarrow \Diamond B)$.
Globally, B is eventually the response to A. If active, when A becomes true the monitor checks that B will become true as well.

Figure 4.2a shows the monitors associated with the absence and universally patterns. Whenever a monitor is in an active state it can check if the invariant is satisfied or violated. Figure 4.2b shows the monitors associated with the precedence and response patterns. For the precedence, the monitor is triggered by the post-condition and later checks violation of the pre-condition. For the response, the monitor is triggered by the pre-condition and later checks violation of the post-condition. All monitors are reset after the invariants have been checked. When there is no label in the transition a monitor maintains the same state.

4.4.2 Shaping

Reward shaping is a technique that allows modifying the rewards given to the agent in some states of the environment so as to help the RL agent to learn more accurately and converge to the goal faster. It provides more guidance to the agent as it explores the environment. However, one has to be careful on how to modify the rewards because this can lead to unexpected consequences. For

example, if we only reward an agent for going in the right direction, the agent could learn to go in circles rather than reach the goal [157]. WiseML utilizes the rewards defined by the designer in order to shape the reward received by the agent during learning. Ultimately, the shaping component will reward the agent for respecting the invariants defined by the monitors and will punish it for violating them.

4.4.3 Enforcing

Monitors can be configured to act as *enforcers*. In this case, they block the action proposed by the agent from being executed in the environment if a violation of the monitored invariant is about to happen. If the violation is not safety-critical a monitor can be configured only to shape the reward in order to help the agent with extra domain knowledge. On the other hand, if the invariants modelled in a monitor are safety-critical, the monitor can enforce them by blocking the dangerous action and executing a safe one. In the current implementation of WiseML the developer can specify a particular action to enforce. If no safe-action is specified and a violation is about to happen the RL agent is asked to produce a new action that does not cause any violation.

Since several monitors can run in parallel, each modelling and possibly enforcing different invariants, the enforcing module has to issue an action that satisfies all invariants of the monitors. At each step, when the agent proposes an action, it is possible that more than one monitor can transition to a *violation* state. The monitoring component communicates the *unsafe actions* of each monitor to the enforcing component that executes the actions that the designer has specified to manage the violation of the invariant.

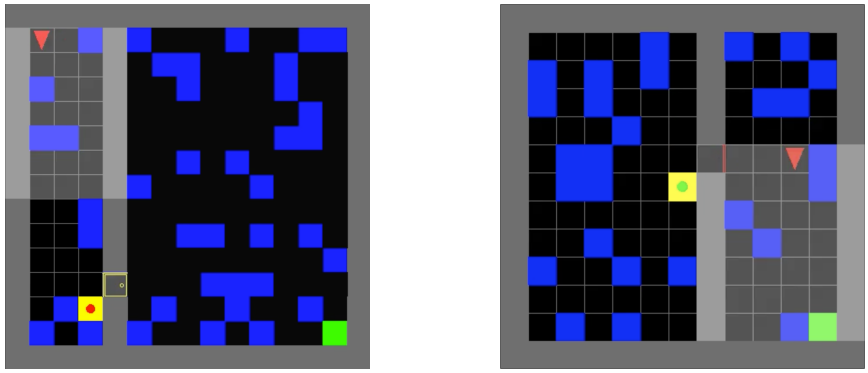
4.5 Evaluation

4.5.1 Gridworld Environment

In order to evaluate our approach, we have designed the WiseML framework and developed an extension of the *Minimalist Gridworld Environment* for OpenAI Gym [134] that supports the WiseML framework. The agent consists of a known-working RL implementation [158]. It is based on a variant of one of the latest RL algorithms developed by Google Deep Mind: *Asynchronous Advantage Actor-Critic* method (A3C) [159]. The algorithm used here is often referred to as A2C since it is a *synchronous* version of the A3C [160].

A gridworld environment consists of a two-dimensional grid of cells. The agent always occupies one cell of the grid facing one of the four adjacent cells. It can interact only with the cell it is currently facing or change direction inside its cell. At each step the agent can choose to perform one of the following actions: *move forward*, *turn left*, *turn right*, *toggle*, and *wait*. The action *toggle* can both open/close a door and turn on/off a light switch.

We have extended the existing grid by introducing new elements in order to evaluate some *safety-critical* scenarios with WiseML. Figure 4.3 shows some examples of randomly generated safety-critical environments. The agent is depicted as a red triangle (top-left corner). The green cell (bottom-right corner) is the agent *goal*, in all the environments the agent has to learn to navigate



(a) An initial configuration. The light is off and the door is closed.

(b) An intermediate step. The light is on and the door is opened.

Figure 4.3: Examples of randomly generated environments.

safely from the initial point of the grid to the goal. The lighter cells around the agent represent its field of view, meaning the observations that agent perceives from the environment. These cells are perceived at each step and semantically analyzed by the perception component. The blue cells (randomly positioned in around all the grid) are *water* cells, if the agent steps on them it drowns and dies, specifically the RL algorithm terminates one episode and the agent starts again from the initial position. There are also more complex elements such as doors and a light switches (yellow tile with a red dot in the middle). By turning on the light-switch next to the door the agent is able to perceive observations in the other room, otherwise, its observations are altered and it can not see potential hazards such as the water. Before reaching the goal cell the agent has to learn to turn on the light by toggling the switch *before* entering a new room.

The environment generation randomly places the wall, door, and n water tiles. The minimum width of each room is two tiles. The light-switch is always placed next to the door. The position of the agent and the goal are fixed to ensure that the agent needs to traverse the entire room to reach the goal. The algorithm also validates that the generated environment has a solution by checking that the goal is reachable from the initial position (e.g., it checks that water cells are not blocking it).

The goal of the agent is to localize the goal position and step on it. However, when the light is off, due to the assumption that the sensors need a minimum amount of light to work, the agent is not able to perceive any observations. Hence, an implicit sub-goal is to turn on the light on before going back to the original goal of reaching its final position.

We have modelled the invariants of the RL agent as LTL properties using the patterns of WiseML. See below a list of some of the conditions used in the formulation of the properties. Each condition is triggered by the perception component of the agent.

Conditions used to detect the *context* of the agent (function of the agent's *perceptions*). (i) p_{wt}: the agent is near the water, (ii) p_{dr}: the agent detects a door in front, (iii) p_{do}: the agent detects that the door is open,

(iv) p_{dc}: the agent detects that the door is closed, (v) p_{lw}: the agent detects light-switch, (vi) p_{lo}: the agent detects that the light is on, (vii), and (viii) p_{lf}: the agent detects that the light is off.

Conditions used to model the monitor invariants of the agent. These are situations that might trigger the monitor (functions of the agent perception and of the *action* proposed by the agent):

— a_{fwt}: the agent moves forward when facing water;
— a_{tgl}: the agent is about to toggle a light-switch.

From the above conditions, we have formulated the following properties, and easily model them as monitors in WiseML:

— (*Absence*) $p_{wt} \implies \Box(!a_{fwt})$. Always avoid to step on water.
— (*Universally*) $p_{lw} \implies \Box(p_{lo})$. The light is always on.
— (*Precedence*) $p_{dr} \implies \Box(!a_{frm} \mathcal{W} p_{lo})$. The light should have been turned on before entering a room.
— (*Response*) $p_{lw} \implies \Box(p_{lf} \rightarrow \Diamond a_{tgl})$. If a light switch is detected and the light is off. Enforced action: *toggle*
— (*Response*) $p_{dr} \implies \Box(p_{dc} \rightarrow \Diamond a_{tgl})$. If a door is detected and the door is closed. Enforced action: *toggle*.

In our experiment each invariant is *enforced* to the agent. It is important to highlight that for the response properties, we implemented the monitor to respond immediately to the pre-condition. Then the post-condition is enforced to happen in the next state. This is a valid implementation since we have the knowledge from the environment that, when the pre-condition is true, it is always possible to perform immediately the actions that satisfy the post-condition.

The developer can choose to specify a particular action to enforce in case of violation or let the agent propose a new suitable action. It is important to notice that the invariants that we have modelled are very simple requirements of the agent and are not related to the task that the agent has to achieve.

4.5.2 Evaluation

In order to evaluate WiseML, we have run the experiments in randomly generated environments of different sizes. Starting from a configuration file we generate the environment and the monitors inside WiseML and run the experiments inside a Docker container.

In each experiment, we have compared the performances of the RL agent with the *safety envelope* of WiseML with another RL agent that uses the exact same RL algorithm and configurations but it is no wrapped with the WiseML framework. For the rest of the paper, we will refer to the WiseML agent as the one with the *safety-envelope* and as SimpleRL agent to the one without.

After modelling the invariants into WiseML, we have collected the results as follows. First we generate a random safety-critical environment `Env` of some size `N`. Then we launch the training of WiseML agent and later the SimpleRL agent from scratch on `Env` until they converge and repeat the process for `M` iterations. Finally, we collect the results of these runs and we start again the training on a different random environment.

The environment is randomly generated regarding the number and position

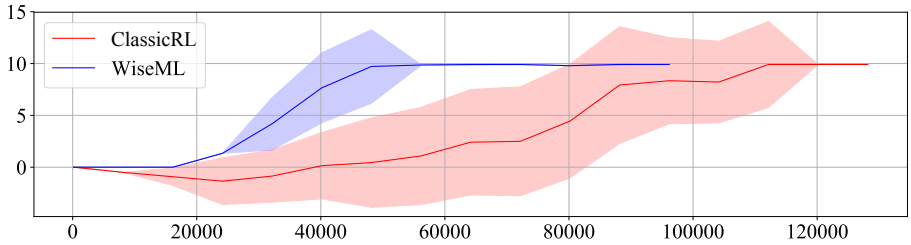


Figure 4.4: Example of one experiment, comparing the convergence with and without WiseML. The Y-axis represents the reward accumulated at each episode by the RL agent. The X-axis represents the number of steps.

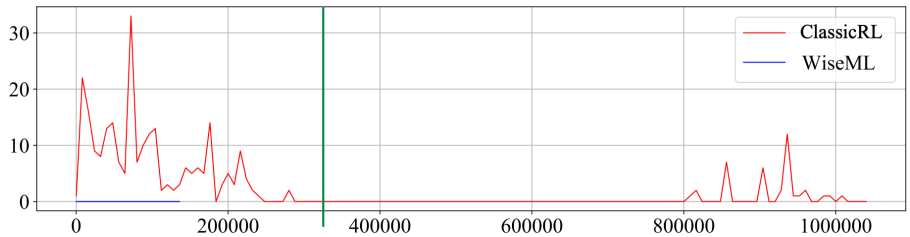


Figure 4.5: Example of one experiment, showing the number of deaths accumulated over time. The SimpleRL agent can continue to die also after convergence (indicated with the vertical green line).

of the water tiles, and the position of the door, light-switch and wall. Regarding the *convergence* of the agents, we consider an agent to have terminated the training when its convergence conditions, described below, are met for several consecutive episodes. An episode terminates when the agent reaches a terminal state, so when it reaches the goal, it dies (e.g. steps on the water) or it reaches a maximum number of steps which is proportional to the size of the grid. The algorithm converges if all the following conditions are satisfied: (i) the goal is achieved, (ii) the number of steps to the goal stabilizes, (iii) the value loss is less than 0.01, and (iv) the mean cumulative reward is positive.

The *value loss* measures the error between the predicted value of a state and the updated value after the reward has been received from the environment.

We have formulated the following research questions:

RQ1 *To what extent* WiseML *can assure the respect of invariants on a reinforcement learning agent using runtime monitoring?*

RQ2 *Can* WiseML *help the agent to converge to its goal faster by combining runtime monitoring with the use of reward shaping?*

To answer the research questions we have modelled the five invariants mentioned in this section as monitors in WiseML.

In our study, we have defined environments (grids) of sizes 7x7, 9x9, 11x11, 13x13, and 15x15. For each size, we have generated 10 random gridworld safety environments, and for each environment, we have performed 10 iterations of the WiseML and SimpleRL agent for a total of *3000 runs* of the RL algorithm until the convergence criteria are met, or the maximum number of steps is reached. Each iteration has a maximum of $size^2 * 10000$ time-steps. The number of water tiles was defined as 25% of the free tiles. The agent receives a positive reward of 10 when it reaches a goal and a negative reward of

Size	Max. number of steps	Convergence (%)	
		WiseML	SimpleRL
7	490000	96.33	78.00
9	810000	91.33	47.67
11	1210000	80.67	33.00
13	1690000	55.33	9.00
15	2250000	66.00	2.00

Table 4.1: Percentage of learning iterations that converged.

Size	Faster (%)	Catastrophes	
		WiseML	SimpleRL
7	45.96	0.00	4483.83
9	38.77	0.00	8301.93
11	41.64	0.00	5970.11
13	41.07	0.00	2663.88
15	54.92	0.00	3053.00

Table 4.2: Comparison between the learning iterations that converged.

-10 for death, -0.1 for violations (only for the WiseML agent) and -0.005 for each step. The agent view of the environment is a grid of size 7x7. We have chosen such coefficient by empirically trying several values and noticing that the agent converged better with these ones. Generally, the worst state is the more negative is the reward.

Table 4.1 shows the percentage of iterations in which the reinforcement learning converged before a predefined maximum number of steps. Table 4.2 shows a comparison between the WiseML and SimpleRL agents for the cases in which the reinforcement learning algorithm converged. The first column shows the average of the comparison in terms of average time-steps between the WiseML agent and SimpleRL agent on the same random environment. In some environments, it was not possible to do the comparison because the SimpleRL agent never converged for such environment. The second and third columns show the average number of catastrophes in each iteration for the WiseML and SimpleRL agent.

Our experiments show that the WiseML agent converges from 55% to 96% of the times, while the SimpleRL agent only converges between 2% to 78% of the times depending on the size of the random environment. Moreover, the WiseML agent is on average faster to converge than the SimpleRL agent. Also, the monitors have prevented catastrophic events to occur (i.e. stepping on the water).

Figure 4.4 and Figure 4.5 show an example of two experiments. Figure 4.4 shows the convergence of WiseML and the SimpleRL agents. It represents the total reward (mean and standard error) accumulated by each agent until convergence, averaged every 8000 steps. Figure 4.5 shows the number of deaths accumulated by the agents over one run; in particular, we show how

the SimpleRL agent can keep dying also long after its convergence (indicated with the vertical green line). Obviously, assuming that the perception layer is perfect, the WiseML agent never died during all the experiments. All our results, including some videos, are available in the link below[2] and they are all reproducible by launching the same experiments via the original code on Github[3] or simply launching them via the Docker image[4].

To answer our research questions, our results show that with WiseML the agent never violates its modelled requirements. Indeed, there is the implicit assumption that the requirements should be correctly modelled using the specification patterns and that the perception component is working correctly. Furthermore, thanks to runtime monitoring and *reward shaping* the agent will converge faster while avoiding catastrophes.

4.6 Conclusions and Future Work

We presented WiseML, an approach that uses runtime monitoring to prevent a RL agent from performing actions that can be dangerous to the environment or to itself. We specified *invariants* through the use of specification patterns, which are translated into *monitors* that block and enforce them. The use of WiseML during learning time improves also the learning of the RL agent. Our approach is agnostic with respect to the chosen RL algorithm and it assumes that the RL agent has no previous knowledge about the environment. Furthermore the agent only has a *partial-observability* of the environment, making it closer to *real-world* applications. We developed and evaluated our approach using one of the latest RL algorithms. We collected data from randomly generated environments and showed how the monitors always enforce their invariants while helping the RL agent to converge to its goal.

As future work, we plan to perform larger experimentation by using other RL algorithms and more complex environments. Moreover, we will investigate more systematic ways of identifying invariants. Finally, we plan to extend our approach by automatically synthesizing all the monitors from the invariants that we want to preserve.

4.7 Acknowledgement

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP).

[2] `https://goo.gl/FzgEdo`
[3] `https://rebrand.ly/wisemlpatterns`
[4] `https://rebrand.ly/wisemldocker`

Chapter 5

Paper D

CROME: Contract-Based Robotic Mission Specication

P. Mallozzi, P. Nuzzo, P. Pelliccione and G. Schneider

International Conference on Formal Methods and Models for System Design. IEEE/ACM, 2020.

Abstract

We address the problem of automatically constructing a formal robotic mission specification in a logic language with precise semantics starting from an informal description of the mission requirements. We present CROME (Contract-based RObotic Mission spEcification), a framework that allows capturing mission requirements in terms of goals by using specification patterns, and automatically building linear temporal logic mission specifications conforming with the requirements. CROME leverages a new formal model, termed Contract-based Goal Graph (CGG), which enables organizing the requirements in a modular way with a rigorous compositional semantics. By relying on the CGG, it is then possible to automatically: i) check the *feasibility* of the overall mission, ii) further *refine* it from a library of pre-defined goals, and iii) *synthesize* multiple controllers that implement different parts of the mission at different abstraction levels, when the specification is realizable. If the overall mission is not realizable, CROME identifies mission scenarios, i.e., sub-missions that can be realizable. We illustrate the effectiveness of our methodology and supporting tool on a case study.

5.1 Introduction

In the near future, service robots will increasingly be used to support tasks in everyday life [48, 161, 162], even though existing solutions are often not readily usable [163]. Service robots are "a type of robot that performs useful tasks for humans or equipment excluding industrial automation applications" [164]. The service robotics market is estimated to reach a value of $24 billion by 2022 [162]. However, the robots that we find in the market today are highly specialized to accomplish a specific function. Their use often reduces to clicking a specific button that will trigger the execution of the specific mission the robot is programmed for. For instance, this is the case of commercial vacuum cleaner robots like Roomba [165]. On the other hand, the advent of multipurpose service robots, required to accomplish various domain-specific missions, calls for new languages and tools to enable end users to accurately specify complex missions [48, 61, 166].

We make a distinction between a *mission requirement*, i.e., an informal description of the mission the robots must perform, and a *mission specification*, i.e., a formulation of the mission in a formal (logical) language with precise semantics [48]. Producing a mission specification from a mission requirement is identified as the *mission specification problem*. Many results in the literature highlight the advantages of specifying robotic missions in a temporal logic language, like linear temporal logic (LTL) or computation tree logic (CTL) [32–45]. Using formal languages makes behavioral specifications precise and unambiguous. However, logic formulas can be difficult to interpret for the end user, and generating them can be an error-prone process [46–48]. Mission requirements, on the other hand, are often ambiguous [54–56] and make it hard to assess the correctness of the specification [57–60]. In recent years, there have been many proposals for describing mission requirements based on: *i)* domain specific languages [61–63], *ii)* natural language [55], and *iii)* visual and end-user-oriented environments [64–67], mostly used for educational purposes.

While the approaches above provide substantial contributions to the mission specification problem, solutions that can scale to complex missions and enable the deployment of service robots in everyday life are still elusive.

As stated in the Multi-Annual Roadmap for Robotics in Europe (MAR) [167], to reduce costs and establish a vibrant component market, we need tools that can support mission *reuse* and diversification, as well as the *variability* of conditions and application scenarios occurring in a real mission. This is also witnessed by our findings during a collaboration with practitioners in the robotic domain [168]. While it is often not difficult to define what the robots should do, the challenge is in coping with the variability of the environments in which the robots operate, especially those involving humans [168]. To address this issue, we would need to explicitly enumerate all the possible variants of a mission using, e.g., state diagrams or flow charts, which can be difficult, tedious, and error-prone.

Menghi et al. [48] have recently identified and proposed a catalogue of robotic movement patterns, which are solutions to recurring problems in mission specification. Patterns are based on LTL formulas, which are often used to automatically synthesize plans [35, 38, 169–171]. While grounded in a formal language, these patterns can also be used by non-experts and there exist tool support to compose them via conjunction or disjunction for the specification of complex missions [172]. Garcia et al. [61] introduce more complex composition rules, including control-flow operators like fall-back (to define alternative strategies when the previous ones fail), exception-handling (to stop the current execution when an exception is raised and then continue with the current task), and sequence (to perform a sequence of tasks). However, control-flow operators are implemented in software but lack a formal representation in logic, which makes it difficult to verify the feasibility of the entire mission specification in a way that is independent of the implementation.

In this paper, we propose a framework, named CROME (Contract-based RObotic Mission spEcification), that explicitly addresses the problems of *specification reuse* and *environment modeling* in mission specification, enabling the designer to cope with the variability of the application scenarios of a robotic mission. By building on recent work on contract-based requirement engineering [14,169,170], leveraging *context-aware contract models* and patterns to generate controller specifications, we decouple the task specification from the specification of the context in which the task is executed. End users explicitly specify the various mission tasks together with their contexts. The overall mission is then automatically compiled by CROME. CROME contributes to the following aspects of the mission specification process:

— *Formulating mission requirements.* We model each requirement as a *goal*, expressed using a set of previously proposed patterns [48, 49]. Goal models have been used over the years as an intuitive and effective means to capture the designer's objectives and their hierarchical structure [86]. In this paper, we augment the notion of goal to explicitly include a concept of *context*, which enables building mission specifications that are adaptable to different environmental conditions. Contexts help capture the variability associated with a mission goal, so that the same goal can be implemented in different ways when used in different contexts.

— *Generating mission specifications.* We introduce a novel model, termed *contract-based goal graph* (CGG), which is automatically generated to formalize a mission and its sub-missions. The CGG is a graph of goals where the root node represents the overall mission, its immediate children represent mission *scenarios*, and the rest of the nodes are part of the sub-missions. In a CGG, goals are captured by assume-guarantee contracts [13] and are linked together using operations and relations between contracts. We differentiate the scenario nodes from other nodes since they are goals that have mutually exclusive contexts and identify sub-missions that cannot be jointly realized.

— *Refining mission specifications out of a library of goals.* We introduce an algorithm that automatically refines the leaf nodes of a CGG using the goals in a library, so that "abstract" goals in the CGG can be further implemented (refined) by more "concrete" goals.

By formalizing the mission specification with a CGG, CROME also offers the following capabilities:

— *Requirement conflict identification.* By checking the satisfiability of the CGG contracts, we are able to identify the presence of conflicts in the mission requirements and immediately inform the designer, before attempting at synthesizing a controller.

— *Realizability checking and controller generation.* CROME checks the realizability of each scenario in the CGG and informs the designer of which sub-goals can be realized (i.e., a controller can be synthesized), given a model of the environment. For each realizable goal of the CGG, CROME synthesizes a controller in the form of a Mealy machine. The controllers are produced together with the CGG.

Our case study shows that the modularity of the CGG allows efficiently checking the feasibility of a mission. The identification of the scenarios allows analyzing the impact of environment variability on the realizability of the robotic mission. The automatic refinement from a goal library facilitates the reuse of existing goals to implement complex specifications. Finally, mutually exclusive scenarios can point to control architectures that may not have a centralized implementation, while still being realizable in a decentralized fashion.

The rest of the paper is organized as follows. In Section 5.2 we provide background notions and related work on contracts, linear temporal logic, specification patterns, and contexts. We introduce CROME in Section 5.3. We detail how the robotic mission is specified and mutually exclusive contexts are generated in Section 5.4 and Section 5.5, respectively. We present the CGG in Section 5.6 and illustrate our approach on a case study motivated by a care center in Section 5.7. Finally, in Section 5.8, we draw some conclusions.

5.2 Background and Related Work

We provide some background on the basic building blocks of CROME: contracts, linear temporal logic, and specification patterns.

5.2.1 Assume-Guarantee Contracts

Contract-based design [13,14] has emerged over the years as a design paradigm capable of providing formal support for building complex systems in a modular way, by enabling compositional reasoning, stepwise refinement, and reuse of pre-designed components.

A *contract* $\mathcal{C}$ is a triple (V, A, G) where V is a set of system *variables* (including, e.g., input and output variables or ports), and A and G are sets of behaviors over V. For simplicity, whenever possible, we drop V from the definition and refer to contracts as pairs of assumptions and guarantees, i.e., $\mathcal{C} = (A, G)$. A expresses the behaviors that are expected from the environment, while G expresses the behaviors that an implementation promises under the environment assumptions. In this paper, we express assumptions and guarantees as sets of behaviors satisfying a logic formula; we then use the formula itself to denote them, with a slight abuse of notation, whenever there is no confusion. An environment E satisfies a contract $\mathcal{C}$ whenever E and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of E are included in the assumptions of $\mathcal{C}$, i.e., when $|E| \subseteq A$, where $|E|$ is the set of behaviors of E. An implementation M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of M are included in the guarantees of $\mathcal{C}$ when considered in the context of the assumptions A, i.e., when $|M| \cap A \subseteq G$.

A contract $\mathcal{C} = (A, G)$ can be placed in saturated form by re-defining its guarantees as $G_{sat} = G \cup \overline{A}$, where $\overline{A}$ denotes the complement of A. A contract and its saturated forms are semantically equivalent, i.e., they have the same set of environments and implementations. Therefore, in the rest of the paper, we assume that all the contracts are expressed in saturated form. A contract $\mathcal{C}$ is *compatible* if there exists an environment for it, i.e., if and only if $A \neq \emptyset$. Similarly, a saturated contract $\mathcal{C}$ is *consistent* if and only if there exists an implementation satisfying it, i.e., if and only if $G \neq \emptyset$. We say that a contract is *well-formed* if and only if it is compatible and consistent. We detail below the contract operations and relations used in this paper.

5.2.1.1 Contract Refinement

Refinement establishes a pre-order between contracts, which formalizes the notion of replacement. Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. $\mathcal{C}$ refines $\mathcal{C}'$, denoted by $\mathcal{C} \preceq \mathcal{C}'$, if and only if all the assumptions of $\mathcal{C}'$ are contained in the assumptions of $\mathcal{C}$ and all the guarantees of $\mathcal{C}$ are included in the guarantees of $\mathcal{C}'$, that is, if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement entails relaxing the assumptions and strengthening the guarantees. When $\mathcal{C} \preceq \mathcal{C}'$, we also say that $\mathcal{C}'$ is an *abstraction* of $\mathcal{C}$ and can be replaced by $\mathcal{C}$ in the design.

5.2.1.2 Contract Composition

Contracts associated with distinct implementations can be combined via the composition operation ($\|$) to specify the composition between the corresponding implementations. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. The

composition $\mathcal{C} = (A, G) = \mathcal{C}_1 \parallel \mathcal{C}_2$ can be computed as follows:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, \tag{5.1}$$

$$G = G_1 \cap G_2. \tag{5.2}$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, hence the operation of intersection in (5.2). An environment for $\mathcal{C}$ should also satisfy all the assumptions, motivating the conjunction of A_1 and A_2 in (5.1). However, part of the assumptions in $\mathcal{C}_1$ may be already supported by $\mathcal{C}_2$ and *vice versa*. This allows relaxing $A_1 \cap A_2$ with the complement of the guarantees of $\mathcal{C}$ [13].

5.2.1.3 Contract Conjunction

Different contracts on a single implementation can be combined using the conjunction operation ($\wedge$). Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. We can compute their conjunction by taking the greatest lower bound of $\mathcal{C}_1$ and $\mathcal{C}_2$ with respect to the renement relation. Intuitively, the conjunction $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ is the weakest (most general) contract that refines both $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}$ can be computed by taking the intersection of the guarantees and the union of the assumptions, that is:

$$\mathcal{C} = (A_1 \cup A_2, G_1 \cap G_2).$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, while being able to operate in either of the environments of $\mathcal{C}_1$ or $\mathcal{C}_2$.

5.2.2 Linear Temporal Logic

Given a set of atomic propositions AP (i.e., Boolean statements over system variables) and the state s of a system (i.e., a specific valuation of the system variables), we say that s *satisfies* p, written $s \models p$, with $p \in AP$, if p is `true` at state s. We can construct LTL formulas over AP according to the following recursive grammar:

$$\varphi := p \mid \overline{\varphi} \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\ \varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$$

where φ, φ_1, and φ_2 are LTL formulas, $\overline{\varphi}$ is the negation of φ, $\vee$ is the logic disjunction, $\mathbf{X}$ is the temporal operator *next* and $\mathbf{U}$ is the temporal operator *until*. Other temporal operators such as *globally* ($\mathbf{G}$) and *eventually* ($\mathbf{F}$) can be derived as follows: $\mathbf{F}\ \varphi = \texttt{true}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi = \overline{\mathbf{F}\ \overline{\varphi}}$. We refer to the literature [16] for the formal semantics of LTL.

5.2.3 Specification Patterns and Context

5.2.3.1 Robotic Patterns

Robotic patterns have been proposed as a solution to recurrent mission specification problems based on the analysis of mission requirements in the robotic literature [48]. CROME supports 22 patterns [48], capturing robot movements and actions performed as a robot move in the environment, organized into three groups: *core movement* patterns, *triggers*, and *avoidance* patterns.

For example, let us assume that the mission requirement is: 'A robot must patrol a set of locations in a certain strict order.' The designer can formulate this requirement by using the *Strict Ordered Patroling* pattern, instantiated for the required set of locations. Let l_1, l_2, and l_3 be the atomic propositions of type *location* that the robot must visit in the given order. The mission requirement can then be reformulated as 'Given the locations l_1, l_2, and l_3, the robot should visit all the locations indefinitely and following a strict order,'[1] leading to the following LTL formulation:

$$\begin{aligned} &\mathbf{G}(\mathbf{F}(l_1 \wedge \mathbf{F}(l_2 \wedge \mathbf{F}(l_3)))) \wedge (\overline{l_2}\ \mathbf{U}\ l_1) \wedge (\overline{l_3}\ \mathbf{U}\ l_2) \\ \wedge\ &\mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_1)) \wedge \mathbf{G}(l_3 \rightarrow \mathbf{X}(\overline{l_3}\ \mathbf{U}\ l_2)) \\ \wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_3)) \\ \wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_2)) \wedge \mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_3)). \end{aligned} \tag{5.3}$$

As shown in this example, a robotic pattern can significantly facilitate the difficult and error-prone task of mission specification.

5.2.3.2 Specification Patterns with Scopes

Our library of patterns is also inspired by the work of Dwyers et al. [49], who developed a catalogue of generic property specification patterns for a broader range of applications. In particular, we adopt the notion of *scope*, which provides a way to define the extent to which a property must hold [49]. For example, for the *universality* pattern, in which we require that a property e be true, we can introduce the following scopes:

$$e \text{ global} = \mathbf{G}(e) \tag{5.4}$$

$$e \text{ before } r = \mathbf{F}(r) \rightarrow (e\ \mathbf{U}\ r) \tag{5.5}$$

$$e \text{ after } q = \mathbf{G}(q \rightarrow \mathbf{G}(e)) \tag{5.6}$$

$$e \text{ between } q \text{ and } r = \mathbf{G}((q\ \wedge\ \overline{r}\ \wedge\ \mathbf{F}\ r) \rightarrow (e\ \mathbf{U}\ r)) \tag{5.7}$$

$$e \text{ after } q \text{ until } r = (\mathbf{G}(q\ \wedge\ \overline{r} \rightarrow ((e\ \mathbf{U}\ r) \mid \mathbf{G}\ p))), \tag{5.8}$$

where q, r are also properties or events. The patterns proposed by Dwyers et al. [49] were also extended to incorporate time [50] and probability [51]. Autili et al. [52] present a unified catalogue of property specification patterns including, among others, the patterns mentioned above [49–51]. A description of the patterns in this catalogue [52] is also available online [53].

5.2.3.3 Context

Many characterizations of the 'context' of an application have been provided, often informally, in the literature. In context-aware ubiquitous computing [173], the context of an application may include information like location, identities of nearby people and objects, time of the day, season, or temperature. More generally, Dey and Abowd [174] define context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an

[1] `http://roboticpatterns.com/pattern/strictorderedpatrolling/`

application, including the user and application themselves." In the robotic domain, Bloisi et al. [175] define mission-related contexts as "choices that are useful in robotic scenarios for adapting the robot behaviors to the different situations." CROME builds on these characterizations and adapts them to robotic missions by formalizing a context as a property associated with a goal and expressed by a logic formula.

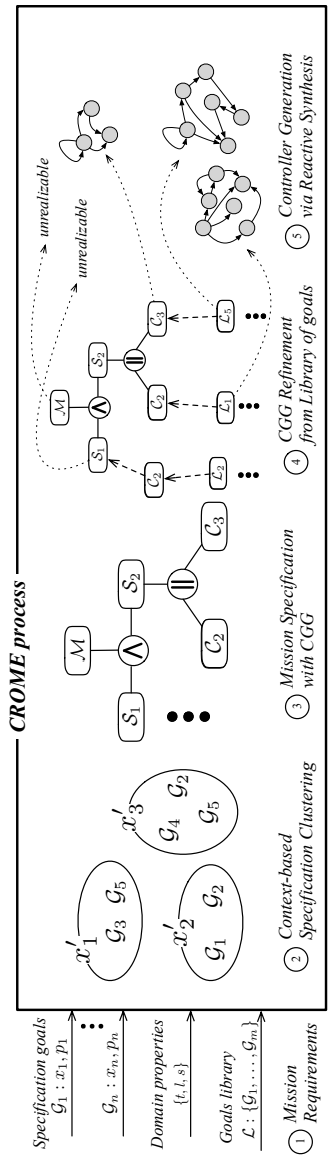


Figure 5.1: Mission specification process

5.3 Overview of CROME

Figure 5.1 shows the mission specification process with CROME, which can be summarized as follows.

Phase (1): A robotic mission can be decomposed into a set of requirements prescribed to a system (the robot) acting in an environment. In the requirement capture phase, the designer provides the inputs to CROME: specification goals, domain properties, and a goal library. Each specification *goal* $\mathcal{G}_i$, modeling a mission requirement, is specified using a *pattern* p_i and instantiated in a *context* x_i. The domain properties encapsulate constraints on the environment and the system. They belong to three categories: (1) physical rules, denoted by t, e.g., specifying the map of a building; (2) logical rules, denoted by l, e.g., specifying logical partitions (areas) of a building; and (3) system constraints, denoted by s, e.g., specifying actions that can not be performed simultaneously by the robotic system, such as going in two locations simultaneously. Finally, the designer can provide a library of predefined goals that will be used in phase (4) to automatically refine the mission specification.

Phase (2): In this phase, CROME analyzes the relationship between goals and groups them in clusters based on their context. For example, two goals are mutually exclusive if the conjunction of their contexts produces a contradiction with respect to the domain properties. CROME produces clusters of goals and associates to each cluster a new generated context x'_j, which is guaranteed to be mutually exclusive with any other cluster's context. This phase is detailed in Section 5.5.

Phase (3): CROME builds the Contract-based Goal Graph (CGG), a formal model representing a graph of goals. CROME formulates *assume-guarantee (A/G) contracts* for each goal and leverages the clusters created in phase (2) to determine the structure of the CGG, which in this phase takes the form of a tree. Contracts belonging to the same cluster are combined using composition; the resulting contracts of the clusters are combined using conjunction. By building the CGG, CROME analyzes the consistency of the mission and helps detect conflicting requirements.

Phase (4): Given a library of goals $\mathcal{L}$, for each leaf node of the CGG, CROME checks whether it can be refined by a goal of the library. Checking refinement between goals amounts to checking refinement between the contracts formalizing the goals. If a library goal is found, it is connected to the CGG and becomes a new leaf node to be scheduled for refinement. The refinement procedure continues recursively until no more library goals can be used to extend the CGG. A library goal can refine multiple goals of the CGG. In this case, we introduce a new leaf for the library goal and connect it with all the goals it refines, which makes the CGG no longer a tree.

Phase (5): CROME checks the realizability of each node of the CGG. If the root of the CGG $\mathcal{M}$ is realizable, then a controller can be generated for the whole mission specified by the designer. If the root node is not realizable, then it may still be possible to realize some of the mission scenarios, as identified in the CGG. Moreover, for each scenario, different controllers can be generated at different abstraction levels. We provide details for this phase in Section 5.6.2.3.

5.4 Capturing Mission Requirements

Mission requirements are provided by the designer in terms of goals and domain properties, expressed in a structured way using patterns and scopes, which will be automatically translated into LTL formulas. Goals and domain properties are defined over a set of atomic propositions AP, which can be true or false at any point during the mission. In the following, we detail the building blocks of the mission requirements, i.e., atomic propositions, contexts, goals, and domain properties.

5.4.1 Atomic Propositions

Atomic propositions (APs) can be grouped into six categories based on the semantics associated with them. *Sensor* APs, *location* APs, and *action* APs are associated with the robot. *Location-context* APs, *time-context* APs, and *identity-context* APs are associated with the context. APs can refer to *controlled* or *uncontrolled* variables for the robot. For example, location APs and action APs refer to controlled variables, since a robot can choose its next location and action, while the other APs refer to uncontrolled variables, since they relate to the context or the perception of the environment.

5.4.2 Context

We formalize *contexts* in terms of Boolean predicates encoding the situation in which a goal must be active. For example, context propositions can encode information related to the location, time, or identities associated with a goal. In a robotic application, locations specify *where* a robot can be, time specifies *when* a certain goal must be active (e.g., during the day or the night), and identities specify the state of external entities (*who*) that may interact with the robot.

5.4.3 Goals

In CROME each mission requirement is modeled by a goal, characterized by the following elements:

- *Name*: goal identifier;
- *Description*: English description of the mission requirement;
- *Context*: Boolean predicates over the context APs that hold true for the goal;
- *Objective*: formulas over all the AP expressing what the robot must achieve under the context of the goal.

Goal objectives can be expressed, for example, by properties including atomic propositions in combination with Boolean operators and the temporal operator **G** (globally), which suits a large number of natural-language requirements [30]. However, CROME enables the expression of more complex objectives. We address the generation of complex temporal logic formulas via the robotic patterns [48] and the specification patterns with scopes [49] in Section 5.2.3.

5.4.4 Domain Properties

Domain properties are general constraints that must hold for the whole mission; they can relate to the robotic agent or the environment and use any type of AP. Domain properties can also be generated by using patterns or basic logic predicates over the APs. CROME accepts three kinds of domain properties, which are all compiled as logic predicates:

— *Mutex properties* relate predicates that cannot be true at the same time. For example, `warehouse` and `shop` AP can be marked as mutex propositions, since they represent separate physical environments where the robot cannot be at the same time.

— *Inclusion properties* express constraints on pairs of propositions, predicates, or patterns, such that, when the first term is true, then also the second term must be true. For example, *SequencedPatroling*(`cashier`, `entrance`, `warehouse`) and *Patroling*(`shop`) can be part of an inclusion property since whenever the robot patrols the `cashier`, `entrance`, and `warehouse` locations in sequence, then it patrols the `shop`.

— *Adjacency properties* express constraints over location APs that are adjacent, i.e., such that one location can be reached within one step from another location. For example, adjacency properties can be used to describe the grid-map of the environment, eventually constraining the movements of the robotic system.

Separating the domain properties from the goals is instrumental to mission specification reuse, as it allows instantiating the same goals in different environments enjoying different domain properties.

5.5 Context-Based Specification Clustering

In this phase, CROME groups the goals into separate *clusters* based on their contexts. If the contexts associated with two goals are jointly satisfiable, then the goals are placed in the same cluster; otherwise they are placed in different clusters, which are marked as mutually exclusive. A cluster is then a tuple

containing a mutex-context and a set of goals.

Algorithm 1: Extract mutually exclusive context clusters

```
Input: goals: list of goals, rules: domain properties
Output: clusters: set of tuples, where each tuple contains a mutex-context and a
        set of goals
goals_cxts ← extract_context(goals)
mtx_cxts ← ∅
/* Compute all the possible combinations for L goals_cxts                    */
for i in {0..L} do
    /* Extract all combinations of i contexts                                */
    comb_i ← combinations(contexts, i)
    /* For each combination                                                  */
    for comb in comb_i do
        for ctx in contexts do
            /* Saturate the combination                                      */
            /* If the context is not part of the combination                 */
            if ctx not in comb then
                /* Get the negation of the context formula                   */
                ctx_neg ← Not(ctx)
                /* Add the negation to the combination                       */
                comb_i ← comb_i ∪ ctx_neg
        /* Add additional rules when needed                                  */
        for r in rules do
            if r applies to comb then
                comb ← comb ∪ r
        /* Simplify formulas in comb (1)                                     */
        comb ← simplify(comb)
        if comb is consistent then
            /* Conjoin all the elements in comb and save the result in
               new_contexts                                                  */
            mtx_cxts ← mtx_cxts ∪ And(comb)
/* Group contexts in mtx_cxts (2)                                            */
mtx_cxts ← group(mtx_cxts)
clusters ← ∅
for cxt in mtx_cxts do
    /* Map goals to contexts in mtx_cxts (3)                                 */
    c_to_g = map(ctx, goals)
    clusters ← clusters ∪ c_to_g
/* Select final clusters (4)                                                 */
clusters ← select(clusters)
return clusters
```

Algorithm 1 automates this phase. It takes as inputs the list of goals and the domain properties, referred as `rules`. The result is a set of clusters, each associated with a new *mutex-context*, which is inconsistent with any mutex-context associated with another cluster. First, the algorithm extracts all the contexts from the list of goals and computes all the possible combinations of contexts. For each combination of contexts `comb` the algorithm performs the following operations:

— *Saturation*: it adds to the combination the negation of all the contexts that are not part of the combination.

— *Adding rules*: if any predicate in `comb` contains APs that are also in a predicate of the `rules`, then it adds the predicate in the `rules` to the combination.

— *Consistency check*: if the conjunction of all the contexts in a combination, after the addition of the `rules` predicate is consistent, meaning that the resulting formula is satisfiable, then it produces a new mutex-context, obtained from the conjunction of the contexts augmented with the `rules`.

Algorithm 1 then groups the mutex-contexts, maps each goal to a mutex-context to form different clusters, and selects the final mutex-contexts among those that are mapped to the same set of goals. Given L contexts, there are at most M combinations, where

$$M = \sum_{k=1,\dots,L} \binom{L}{k} = \sum_{k=1,\dots,L} \frac{L!}{k!(L-k)!}.$$

Each combination contains at most $N = L + R$ elements in conjunction, where L is the number of contexts and R is the number of additional rules, i.e., domain properties related to the context. For example, Figure 5.2 shows a list of context combinations that are mapped to a list of goals. Each combination is formed by propositions x_{ij} representing contexts and rules, and their conjunction results in a mutex-context x'_i. Every mutex-context x'_i is then associated with a set of goals $\mathcal{G}$ to form a cluster. We detail below some of the functions used in Algorithm 1.

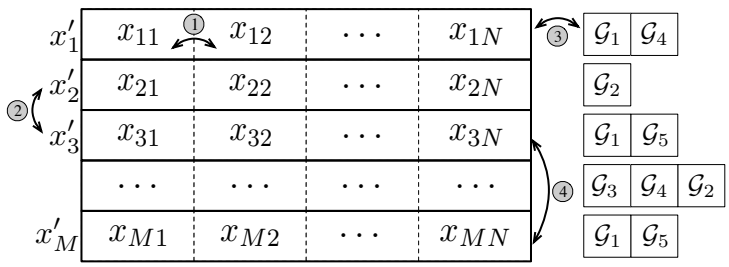


Figure 5.2: Example of mapping of 5 goals $\mathcal{G}_1, \dots, \mathcal{G}_5$ to M mutex-contexts x'_1, $\dots$, x'_M. Each combination of contexts contains at most N logic propositions in conjunction. A circled number indicates refinement checking tasks among formulas expressing the goal contexts or mutex-contexts.

In CROME, contexts and mutex-contexts are formulas. Therefore, to manipulate contexts, we define a refinement relation between formulas. A formula ϕ refines a formula ψ if and only if $\phi \rightarrow \psi$ is valid. If ϕ refines ψ, then the behaviors satisfying ϕ are included in the set of behaviors satisfying ψ. The *simplify*, *group*, *map*, and *select* functions in Algorithm 1 perform refinement checks among pairs of LTL formulas, as also marked by the circled numbers in Figure 5.2:

— (1) ***simplify***. Each mutex-context is built as a conjunction of clauses. Each clause x_i represents a context, a negation of a context, or a context rule. For each pair of clauses x_{ia}, x_{ib} in a mutex-context x'_i, Algorithm 1 checks whether x_{ia} refines x_{ib} and removes the most abstract clause, e.g., if $x_{ia} \rightarrow x_{ib}$, then $x_{ia} \wedge x_{ib} = x_{ia}$.

— (2) ***group***. This process is similar to the one in *simplify*, but operates on pairs of mutex-contexts x'_i, x'_j, as shown in Figure 5.2, rather than the clauses of each mutex-context. For each pair of mutex-context formulas, if a formula implies another one, the group function only retains the most refined one.

— (3) ***map***. The mapping process connects each specification goal to a new mutex-context x'_i, and finally forms a cluster. Let x_i be the context of the specification goal $\mathcal{G}_i$. Then, the map function checks whether all the behaviors satisfying the mutex-context x'_i are contained in context x_i, that is, whether x'_i is a refinement of x_i. If this is the case, CROME links $\mathcal{G}_i$ to the new context x'_i. Because mutex-contexts are constructed by refining contexts, there must exist a mutex-context x'_i that refines x_i.

— (4) ***select***. It may happen that more than one mutex-context are linked to the same goal. For example, in Figure 5.2, both x'_3 and x'_M are linked to $\mathcal{G}_1$ and $\mathcal{G}_5$. In this case, CROME maps the goal to the cluster with the most abstract context.

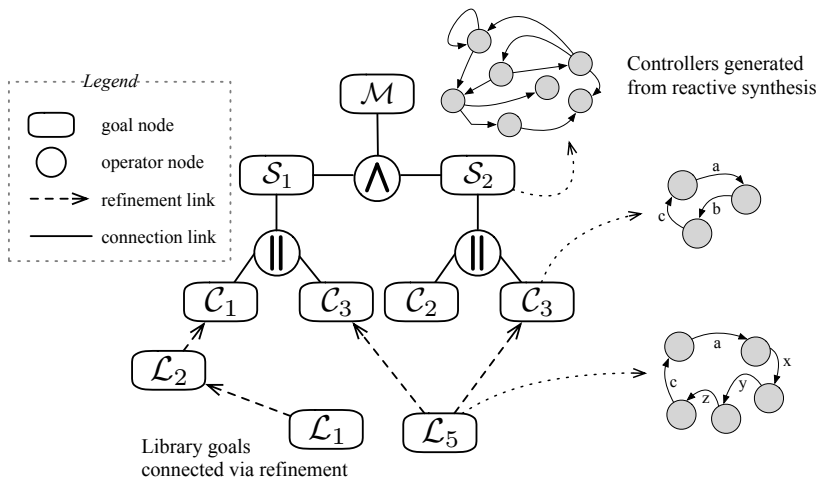


Figure 5.3: Example of CGG where some of the goal nodes are linked to a Mealy machine representing the controller synthesized from the node.

5.6 Mission Specification via Contract-Based Goal Graphs

In this step, domain properties and goals are formalized using A/G contracts and organized using a CGG.

5.6.1 Contract Formalization and Analysis

Once the clusters and mutex-contexts are identified by Algorithm 1, CROME produces one contract $\mathcal{C}_i$ for each goal $\mathcal{G}_i$ in the clusters, where:

— the assumptions capture the domain properties related to the environment in which the mission is deployed;

— the guarantees capture the properties associated with the goal objectives and the corresponding context via formulas of the form

$$\mathbf{G}(ctx \rightarrow obj) \tag{5.9}$$

for a context ctx and an objective obj expressed, for example, as a conjunction of robotic patterns.

Since contract assumptions and guarantees are expressed by logic formulas, CROME checks for incompatibility and inconsistency (i.e., emptiness of assumptions or guarantees) by checking whether the logic formulas are satisfiable. Moreover, CROME performs a *feasibility check* to verify whether contracts are well-formed, i.e., whether $A \cap G \neq \emptyset$ holds. For example, the contract $\mathcal{C} = (\phi_a, \phi_g)$, where $\phi_a := \mathbf{G}(\texttt{env})$, $\phi_g := \mathbf{G}(\texttt{env}) \rightarrow (\mathbf{F}\texttt{move} \wedge \overline{\mathbf{F}\texttt{move}})$, and `env` and `move` are APs, is compatible and consistent. However, $\mathcal{C}$ is not well-formed since $\phi_a \wedge \phi_g$ is infeasible. LTL satisfiability checks can be reduced to model checking problems [14, 176]. We check the satisfiability of a formula ϕ by querying a model checker for the validity of $\psi := \neg\phi$. If ψ is valid, then ϕ is unsatisfiable. A counterexample invalidating ψ is a model, i.e., a satisfying trace, for ϕ.

5.6.2 Contract-Based Goal Graph

A CGG, shown in Figure 5.3, is a graph $T = (\Upsilon, \Sigma)$, where each node $\upsilon \in \Upsilon = \Gamma \cup \Delta$ is either a *goal node* $\gamma \in \Gamma$ or an *operator node* $\delta \in \Delta$, with $\Gamma \cap \Delta = \emptyset$. Each goal node is the formalization of a goal via a contract. Each operator node takes a value in $\{\|, \wedge\}$ and represents an operation (composition or conjunction) between contracts. Each edge $\sigma \in \Sigma$ can connect a goal node in Γ to an operator node in Δ or two goal nodes. In the former case, the edge is a *connection link*. Otherwise, it is a *refinement link*.

Any goal node of the CGG can be realized to achieve a controller. A realization of the root node covers the whole mission but the synthesis problem could be infeasible or intractable in practice. The decomposition of the goals via the modularity of the CGG allows pointing out portions of the mission that may be independently realizable.

5.6.2.1 Building the CGG via Composition and Conjunction

CROME uses contract conjunction and composition to combine the different goals and form the CGG. Composition and conjunction produce more complex goals from simpler ones, which are then connected to the CGG. However, composition demands that the resulting goal operate in environments satisfying the assumptions of all the composing goals. On the other hand, conjunction requires that the resulting goal operate with environments that satisfy either (but not necessarily both) of the assumptions of the original goals. CROME uses contract conjunction to blend different *scenarios* that must be both satisfied by the mission, while the scenarios are built by composition of smaller contracts. More specifically, the CGG is built by computing, for each cluster,

the composition of the goals associated with the cluster (e.g., $\mathcal{C}_2$ and $\mathcal{C}_3$ in Figure 5.3). Goals can be interconnected to form complex structures, where the guarantees of one (or more) goal are used to discharge the assumptions of other goals. Such a composition produces a new goal node (e.g., $\mathcal{S}_1$ in Figure 5.3), which is the scenario supported by the goals in the cluster. Finally, the overall mission specification $\mathcal{M}$ is compiled in terms of the conjunction of all the mutually exclusive scenarios (e.g., $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 5.3).

5.6.2.2 Extending the CGG via Refinement from Library of Goals

A library of goals is a collection of goals, each formalized with a contract and labeled with a *cost*. CROME can automatically extend a CGG by refining its leaf nodes with goals chosen from a library of goals, while minimizing the overall cost. Figure 5.3 shows how contracts $\mathcal{C}_1, \mathcal{C}_2$, and $\mathcal{C}_3$ are refined by the library goals $\mathcal{L}_1, \mathcal{L}_2$, and $\mathcal{L}_5$. A refinement link between a library goal $\mathcal{L} = (a_l, g_l)$ and a leaf node $\mathcal{C} = (a_c, g_c)$ is created if and only if $\mathcal{L} \preceq \mathcal{C}$. $\mathcal{L}_2$ is further refined by library goal $\mathcal{L}_1$, while $\mathcal{L}_5$ refines two goal nodes of the CGG, which are both corresponding to contract $\mathcal{C}_3$.

5.6.2.3 Controller Synthesis

CROME checks the realizability of each goal node of the CGG and, if it is realizable, it produces a controller using reactive synthesis. In Figure 5.3, goal $\mathcal{C}_3$ is refined by $\mathcal{L}_5$ and they can both be realized with two controllers at different abstraction levels. For example, let us assume that $\mathcal{C}_3$ requires as objective *Patroling*(a, b, c), corresponding to the LTL formula $\phi_a = \mathbf{GF}\ a \wedge \mathbf{GF}\ b \wedge \mathbf{GF}\ c$. On the other hand, the library goal objective is *SequencedPatroling*(a, x, y, z, c) corresponding to $\phi_r = \mathbf{GF}(a \wedge \mathbf{F}(x \wedge \mathbf{F}(y \wedge \mathbf{F}(z \wedge \mathbf{F}\ c))))$. Further, there exists a domain property of type inclusion between *SequencedPatroling*(x, y, z) and *Patroling*(b), that is, $\psi = \mathbf{G}(\mathbf{F}(x \wedge \mathbf{F}(y \wedge \mathbf{F}\ z))) \rightarrow \mathbf{GF}\ b$. Given $\mathcal{C}_3 = (\psi, \phi_a)$ and $\mathcal{L}_5 = (\mathit{true}, \phi_r)$, we have and $\mathcal{L}_5 \preceq \mathcal{C}_3$, since $\psi \rightarrow \mathit{true}$ and $\phi_r \rightarrow (\psi \rightarrow \phi_a)$ are valid formulas.

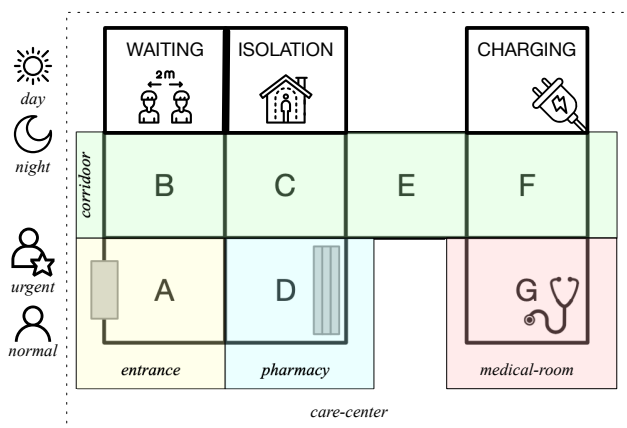


Figure 5.4: Care clinic map showing the time, location, and identity contexts (written in italics).

5.7 Case Study: Urgent Care

We consider a mission performed by a service robot working in an urgent care clinic, and consisting of several tasks. Figure 5.4 shows the map of the clinic together with the contexts. Time contexts (*day*, *night*) and identity contexts (*urgent*, *normal*) capture the variability in the time of the day and type of patient that needs attention. Location contexts (*corridor*, *entrance*, *pharmacy*, *medical-room*, and *care-center*) denote one or more physical locations on the map (A, B, C, D, E, F, G, WAITING, ISOLATION, and CHARGING). Both during the day and during the night, the robot must patrol the clinic. By patroling we mean that the robot should recurrently visit all the rooms (in any order) without letting any room unvisited. To express this property we use the patroling pattern [48].

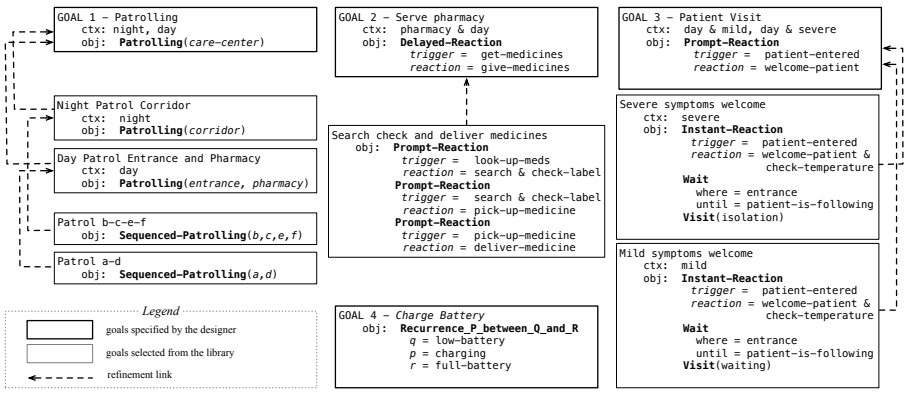


Figure 5.5: Care clinic main goals specified by the designer connected via refinement to the goals selected from the library.

During the day, and when inside the pharmacy, the robot must get the medicine, whenever asked to do so, and give it to the client. It must also welcome new patients at the entrance of the shop. Finally, it must always go and charge the battery when the power level is low. CROME relies on the model checker NuSMV [177] to perform all the checks in the CGG and on Strix[2] to generate controllers via reactive synthesis.

Figure 5.5 shows the mission requirements formalized as goals by the designer and the goals that are selected from the library of goals by CROME to refine the mission requirements. A goal is composed of (i) name, (ii) description, (iii) context, and (iv) objective (Section 5.4). For brevity, in Figure 5.5, we have omitted the goal descriptions. For example the name of the first goal is `Patrolling`, the context is described by two atomic propositions, `night` and `day`, and the objective is an instantiation of the patroling[3] pattern (i.e., instantiated on the AP $\texttt{care} - \texttt{center}$). We use multiple Boolean propositions separated by comma to denote multiple goals, each instantiated in a single context. Specifying *Goal 1* with context $night, day$ is then equivalent to specifying two separate goals for *night* and *day*, respectively. When the context

[2] `https://strix.model.in.tum.de/`

[3] `http://roboticpatterns.com/pattern/sequencedpatrolling/`

is omitted from a goal, then we imply that the goal objectives must hold in all contexts. We also note that *Goal 4* instantiates a property specification pattern, the recurrent pattern,[4] which also uses the scope `between Q and R`.

We observe that *Goal 1* and *Goal 3* are refined differently according to the contexts specified by the designer. In *Goal 1*, patroling is achieved by patroling the *corridor* during the night, while patroling the *entrance* and the *pharmacy* are required during the day. Similarly, for *Goal 3*, the task of welcoming a new patient is refined differently according to the gravity of the symptoms where, in the *severe* cases, the robot visits the isolation room while, in the *normal* cases, it goes in the waiting room.

CROME produces a CGG with a total of 17 goals, 5 of which are the identified scenarios and 11 are library goals that can be reused to refine the leaves of the CGG. Figure 5.6 shows the name of the goals associated to each of the five mutex-contexts. The full example is available online.[5] The root node of the CGG can not be realized by the synthesizer, which runs out of memory. However, every scenario of the CGG is realizable with a maximum and minimum synthesis time of 962.98 s and 24.59 s, respectively. On the other hand, individual nodes can take up to 0.72 s to be realized into a controller. Overall, this example shows how CROME facilitates the formalization of mission requirements, can identify mission scenarios that can be independently realizable, and can refine leaf goals from a library of goals.

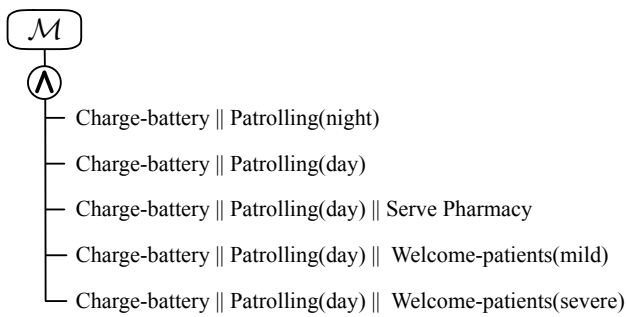


Figure 5.6: Goals belonging to each identified scenario of the case study.

5.8 Conclusions

In this paper we introduced CROME, a design framework for capturing and formalizing robotic mission requirements. CROME facilitates the translation of informal requirements in terms of goals by leveraging a set of specification patterns. It then formalizes the goals in terms of assume-guarantee contracts and leverages a novel, modular representation, namely, a contract-based goal graph (CGG), to analyze the mission specification and detect inconsistencies. Given the CGG, CROME can automatically refine the leaf nodes with goals from a predefined library. It can automatically check the realizability of the overall mission and synthesize a controller, if the mission is realizable. Finally,

[4] `http://ps-patterns.wikidot.com/recurrence-property-pattern`
[5] `https://rebrand.ly/cromeresults`

if the mission is not realizable, CROME can help debug the specification by identifying subsets of requirements in terms of mission scenarios that may be realized. Future work includes devising heuristics to improve on the scalability of the clustering algorithm and further experimentation, also in collaboration with our industrial partners, to investigate the feasibility and usability of the approach in practical and industrial contexts.

Acknowledgments

This work was supported in part by the Wallenberg Al Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and the EU H2020 Research and Innovation Program under GA No. 731869 (Co4Robots). In addition, the authors gratefully acknowledge the support by the US National Science Foundation (NSF) under Awards 1846524 and 1839842, the US Defense Advanced Projects Agency (DARPA) under Award HR00112010003, the US Office of Naval Research (ONR) under Award N00014-20-1-2258, Raytheon Technologies Corporation, and the Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by the Italian Government under CIPE resolution n. 70/2017 (Aug. 7, 2017). The views, opinions, or findings contained in this article should not be interpreted as representing the official views or policies, either expressed or implied, by the US Government. This content is approved for public release; distribution is unlimited.

Chapter 6

Paper E

Incremental Refinement of Goal Models with Contracts

P. Mallozzi, P. Nuzzo, P. Pelliccione

International Conference on Fundamentals of Software Engineering. 2021.

Abstract

Goal models and contracts offer complementary approaches to requirement analysis. Goal modeling has been effectively used to capture designer's intents and their hierarchical structure. Contracts emphasize modularity and formal representations of the interactions between system components. In this paper, we present CoGoMo (Contract-based Goal Modeling), a framework for systematic requirement analysis, which leverages a new formal model, termed *contract-based goal tree*, to represent goal models in terms of hierarchies of contracts. Based on this model, we propose algorithms that use contract operations and relations to check goal consistency and completeness, and support incremental and hierarchical refinement of goals from a library of goals. Model and algorithms are implemented in a tool which enables incremental formalization and refinement of goals from a web interface. We show the effectiveness of our approach on an illustrative example motivated by vehicle platooning.

6.1 Introduction

Missing or erroneously formulated requirements can have a negative impact on the quality of a design. Designers are often faced with the challenge of ensuring the correctness of an implementation despite the growing complexity of the requirement corpora [30]. Existing requirement-management tools are mostly based on natural-language constructs that leave space for ambiguities, redundancies, and conflicts [178, 179]. Furthermore, the requirement elicitation process is itself challenging, as it revolves around human-related considerations that are intrinsically difficult to capture.

Goal modeling (e.g., as in KAOS [28, 180]) has been used over the years as an intuitive and effective means to capture the designer's intents and their hierarchical structure. The refinement process, however, mostly follows informal procedures, e.g., by posing *how* questions about existing high-level goals (top-down process) or *why* questions about low-level goals for the system under consideration (bottom-up process) [29, 30]. The main modeling challenges are framed in terms of ensuring *completeness* and *consistency* of a specification. A set of hierarchically organised goals is *incomplete* when the high-level goal remains unsatisfied even if the low-level goals, which are expected to capture its decomposition, are satisfied, meaning that the designer could not anticipate all the possible operating scenarios for the design. There is, instead, a *conflict* when the satisfaction of a goal prevents the satisfaction of another goal [86]. The process of completely refining a goal into sub-goals is not straightforward [181]. On the other hand, independently-developed goals can include overlapping and conflicting behaviours [182]. Systematic methods to detect conflicts and incomplete requirements remains an active research area [183–185].

This paper presents a framework, CoGoMo (Contract-based Goal Modeling), which addresses these challenges by representing goals via contract models. Contract-based modeling has shown to enable formal requirement analysis in a modular way, rooted in sound representations of the system semantics and decomposition architecture [13, 14, 169, 171, 179, 186, 187]. A *contract* specifies the behavior of a component by distinguishing the responsibilities of the component (*guarantees*) from those of its environment (*assumptions*). Contract operations

and relations provide formal support for notions such as stepwise *refinement* of high-level contracts into lower-level contracts, *compositional reasoning* about contract aggregations, and *reuse* of pre-designed components satisfying a contract. CoGoMo addresses correctness and completeness of goal models by formulating and solving contract consistency and refinement checking problems. Specifically, the contributions of the paper can be summarised as follows:

— A novel formal model, namely, *contract-based goal tree (CGT)*, which represents a goal model as a hierarchy of assume-guarantee (A/G) contracts.

— Algorithms that exploit the CGT as well as contract-based operations to detect conflicts and perform complete hierarchical refinements of goals. Specifically, we introduce mechanisms that help resolve inconsistencies between goals during refinement and a *goal extension* algorithm to automatically refine the CGT using new goals from a library.

— A tool, which implements the proposed model and algorithms to incrementally formalize and refine goals via an easy-to-use web-interface.

We illustrate the applicability of CoGoMo on a case study motivated by vehicle platooning.

6.2 Background

Goals. A goal is a prescriptive statement of intent that the system should satisfy, formulated in a declarative way. Goals can be decomposed, progressing from high-level objectives to fine-grained system prescriptions [28]. For example, an AND-refinement link relates a goal to a set of sub-goals. The parent can be satisfied if all the sub-goals in the refinement are also satisfied. Establishing *correctness* of the refinement amounts to ensuring that the sub-goals are *consistent*, i.e., there are no *conflicts* among them, and *complete*, i.e., there are no behaviours left unspecified that could result in a violation of the high-level goal even if the lower-level goals are satisfied. We only refer to *internal completeness*, i.e., we are not concerned with investigating whether all the information required to define a design problem is in the specification [31]. Formally, we say that the refinement of goal $\mathcal{G}$ into sub-goals $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ is correct if and only if

$$\underbrace{\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \not\models \textbf{false}}_{\text{consistency}} \quad \wedge \quad \underbrace{\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \models \mathcal{G},}_{\text{completeness}}$$

where we denote by $\models$ the entailment operator between goals and say that $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ entails $\mathcal{G}$ to mean that, if all $\mathcal{G}_1, \ldots, \mathcal{G}_n$ are satisfied then $\mathcal{G}$ is satisfied. Similarly, we write $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\} \not\models \textbf{false}$ to indicate that the logical conjunction of $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\}$ does not lead to false.

Contracts. A *contract* $\mathcal{C}$ is a triple (V, A, G) where V is a set of *variables*, and A and G are sets of behaviors over V. For simplicity, whenever possible, we drop V from the definition and refer to contracts as pairs of assumptions and guarantees, i.e., $\mathcal{C} = (A, G)$. A expresses the behaviors that a system expects from its environment, while G expresses the behaviors that a system implementation promises under the environment assumptions. An environment

E satisfies a contract $\mathcal{C}$ whenever E and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of E are included in the assumptions of $\mathcal{C}$, i.e., when $|E| \subseteq A$, where $|E|$ is the set of behaviors of E. An implementation M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of M are included in the guarantees of $\mathcal{C}$ when considered in the context of the assumptions A, i.e., when $|M| \cap A \subseteq G$. A contract $\mathcal{C} = (A, G)$ can be placed in saturated form by re-defining its guarantees as $G_{sat} = G \cup \overline{A}$, where $\overline{A}$ denotes the complement of A. A contract and its saturated forms are semantically equivalent, i.e., they have the same set of environments and implementations. Therefore, in the rest of the paper, we assume that all the contracts are expressed in saturated form [13].

We say that a contract is *well-formed* if and only if it is *compatible*, i.e., $A \neq \emptyset$ and *consistent*, i.e., $G \neq \emptyset$, that is, if and only if there exists at least an environment and an implementation that satisfy the contract. Contract *refinement* formalizes a notion of substitutability among contracts. Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. $\mathcal{C}$ refines $\mathcal{C}'$ if and only if all the assumptions of $\mathcal{C}'$ are contained in the assumptions of $\mathcal{C}$ and all the guarantees of $\mathcal{C}$ are included in the guarantees of $\mathcal{C}'$, that is, $\mathcal{C} \preceq \mathcal{C}'$ if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement entails relaxing the assumptions and strengthening the guarantees. If $\mathcal{C} \preceq \mathcal{C}'$, we also say that $\mathcal{C}'$ is an *abstraction* of $\mathcal{C}$ and can be replaced by $\mathcal{C}$.

Contracts can be combined through the operations of *composition* and *conjunction*. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. The composition $\mathcal{C}_{\|} = (A, G) = \mathcal{C}_1 \parallel \mathcal{C}_2$ can be computed using the following expressions: $A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}$ and $G = G_1 \cap G_2$. The conjunction $\mathcal{C}_{\wedge} = \mathcal{C}_1 \wedge \mathcal{C}_2$ can instead be computed by taking the intersection of the guarantees and the union of the assumptions, that is, $\mathcal{C}_{\wedge} = (A_1 \cup A_2, G_1 \cap G_2)$. Intuitively, an implementation satisfying $\mathcal{C}_{\|}$ or $\mathcal{C}_{\wedge}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, hence the operation of intersection. The situation is different for the environments. Composition requires that an environment satisfy the assumptions of both contracts, motivating the conjunction of A_1 and A_2. On the other hand, contract conjunction requires that an implementation operate under all the environments of $\mathcal{C}_1$ and $\mathcal{C}_2$, motivating the disjunction of A_1 and A_2.

In the following, we denote by $\mathcal{C}_i = (\psi_i, \phi_i)$ the contract formalizing a goal $\mathcal{G}_i$, where ψ_i and ϕ_i are logic formulae used to represent the assumptions and the guarantees, respectively. Finally, we perform operations among goals by translating them into operations on the corresponding contracts.

6.3 Running Example: Vehicle Platooning

We consider a case study inspired by vehicle platooning as an illustrative example throughout the paper. We define goals for a vehicle participating in a platoon in the *following mode*, which adjusts speed and steering angle based on what is communicated by the leading vehicle. We assume that all the vehicles in the platoon communicate via Vehicle Ad hoc Networks (VANETs), established with the IEEE 802.11p standard, a specially designed protocol for intelligent transportation systems (ITS) [188], offering at most 27 Mbps of

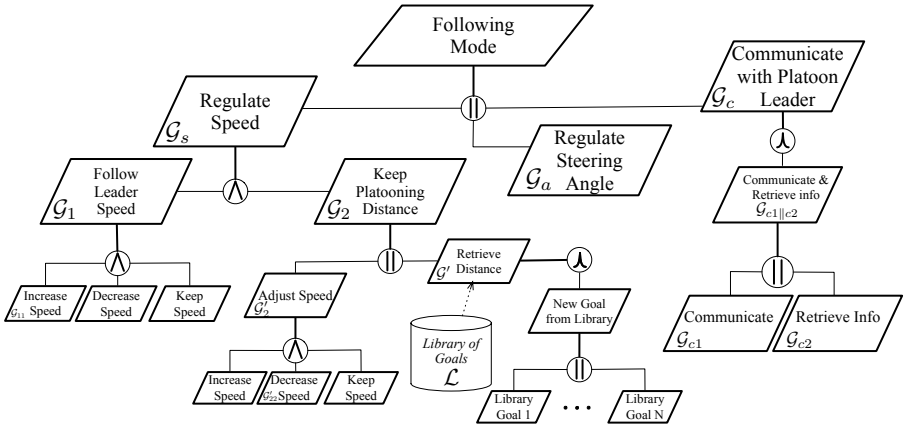


Figure 6.1: Portion of the CGT for the vehicle platooning example.

data transmission rate. The propagation delay is the difference between the time-stamps for message reception, t_{Rx}, and message transmission, t_{Tx}, i.e., $d = t_{Rx} - t_{Tx}$. The necessary time interval $d_{(i,j)}$ for a successful end-to-end transmission of a message of L bits between a pair of vehicles (i, j) is:

$$d_{(i,j)} = \frac{L}{f_{(i,j)}}, \tag{6.1}$$

where $f_{(i,j)}$ is the *transmission rate* between the i-th and the j-th vehicle. We consider a platoon consisting of N vehicles, where the first one is the leader and the remaining $N-1$ are the followers. To reach all followers, a message generated by the leader propagates through for at most $N-1$ hops. We assume the same transmission rate $f = 3$ Mbps between adjacent vehicles and a fixed message length size $L = 400$ bytes.

6.4 The CoGoMo Approach

CoGoMo revolves around a new formal model, termed *contract-based goal tree* (CGT), and a set of operations on it. A CGT is a tree $T = (\Upsilon, \Sigma)$, where each node $\upsilon \in \Upsilon = \Gamma \cup \Delta$ is either a *goal node* $\gamma \in \Gamma$ or an *operator node* $\delta \in \Delta$, with $\Gamma \cap \Delta = \emptyset$. Each goal node contains the formalization of a goal in terms of a contract. Each operator node assumes a value in the set $\{\|, \wedge, \curlywedge\}$ of available operators, namely, composition, conjunction, and refinement, respectively. Each edge $\sigma \in \Sigma$ connects a goal node in Γ to an operator node in Δ or *vice versa*. Figure 6.1 shows a portion of a CGT for the vehicle platooning example, where each goal node includes a textual description of a goal. A *library* $\mathcal{L}$ of goals, at the bottom of the figure, is used to automatically extend the CGT. A library of goals is a set of pre-defined goals, e.g., specification patterns, that can be labelled by a *cost*, capturing the overhead incurred when employing a certain goal to extend the CGT, as further illustrated below.

A CGT can be built interactively using a web-interface and with the support of a proof-of-concept tool, which is released open-source [189]. A designer can insert (new) goals by typing or uploading a structured text file.

The specification formalization process then iterates between two activities: (i) goal identification and formalization with A/G contracts, and (ii) goal manipulation, incrementally linking goals via composition, conjunction, and refinement. The outcome is a formal specification in terms of a CGT.

6.4.1 Goal Formalization

CoGoMo enables requirement formalization by representing goals in terms of contracts. It then solves contract verification problems to detect conflicts and incompleteness among goals. Specifically, completeness and consistency checking problems translate into checking the satisfiability and validity of logic formulas via an SMT solver [190]. In this paper, we express contract assumptions and guarantees as formulas in propositional logic, where atomic propositions include Boolean variables or arithmetic predicates on real variables.

6.4.1.1 Detecting Conflicts.

CoGoMo detects conflicts by checking the compatibility, consistency, and feasibility of each contract formalizing a goal in the CGT. *Feasibility* checking aims to verify that there exists at least an implementation which does not violate the assumptions, that is, for contract $\mathcal{C} = (A, G)$, $A \cap G \neq \emptyset$ holds. CoGoMo verifies compatibility, consistency, and feasibility of a contract (ψ, ϕ) by checking whether ψ, ϕ, and $\psi \wedge \phi$ are satisfiable, respectively. In case of conflict, the SMT solver provides an explanation of infeasibility in terms of an *unsat core*, i.e., a subset of clauses that are mutually unsatisfiable [190]. CoGoMo then links the conflicting clauses to the goals that generated them and presents these goals to the designer.

6.4.1.2 Checking Completeness.

CoGoMo checks completeness by verifying that all the refinement links of the CGT are correct. Given two contracts, $\mathcal{C} = (\psi, \phi)$ and $\mathcal{C}' = (\psi', \phi')$, $\mathcal{C} \preceq \mathcal{C}'$ holds if and only if $\psi' \rightarrow \psi$ and $\phi \rightarrow \phi'$ are *valid* formulas, i.e., they are tautologies for the language, where $\rightarrow$ denotes the implication. Validity checking can be translated into checking the satisfiability of $\overline{\psi' \rightarrow \psi}$ or $\overline{\phi \rightarrow \phi'}$. If no solution is found, then the refinement is correct; otherwise, the returned solution serves as a certificate of incompleteness, and is exhibited to the designer.

6.4.2 Goal Manipulation via Composition and Refinement

CoGoMo uses composition to capture with a single goal the composite behaviors resulting from the composition of modules (implementations) that are separately specified by different goals. For example, goal $\mathcal{G}_c$ in Figure 6.1 can be refined by the composition of $\mathcal{G}_{c1}$ with $\mathcal{G}_{c2}$. Figure 6.2 proposes an initial formalization of $\mathcal{G}_c$ as a contract $\mathcal{C}_c$ and its further decomposition into two goals, $\mathcal{G}_{c1}$ and $\mathcal{G}_{c2}$, establishing requirements on the network connection and the follower's speed and angle ranges, respectively. $\mathcal{G}_{c1}$ specifies the propagation delay d according to the transmission rate f (in Mbps), the message length L, and the position n of the follower participating in the platoon. Assuming a working network connection, $\mathcal{G}_{c2}$ guarantees that the speed of the follower is at

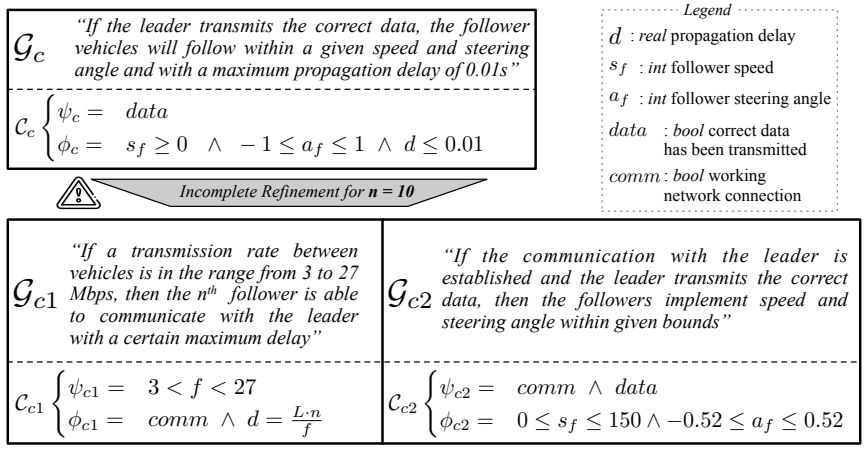


Figure 6.2: Examples of three goals formalized by contracts. After linking the goals via composition and refinement, CoGoMo detects that the refinement is incomplete.

most 150 km/h and its steering angle at most 30° (0.52 rad). We would like to show that $\mathcal{G}_{c1} \parallel \mathcal{G}_{c2} = \mathcal{G}_{c1\parallel c2}$ can be connected to the top goal $\mathcal{G}_c$ via refinement, i.e., $\mathcal{G}_{c1\parallel c2} \preceq \mathcal{G}_c$. To do so, CoGoMo first checks for conflicts between $\mathcal{G}_{c1}$ and $\mathcal{G}_{c2}$ by verifying that the contract associated with $\mathcal{G}_{c1\parallel c2}$ is compatible, consistent, and feasible, which is the case in our example. However, CoGoMo detects that $\mathcal{G}_{c1\parallel c2}$ is not a refinement of $\mathcal{G}_c$ and provides a certificate showing that, for the 10^{th} follower in the platoon, the propagation delay is slightly larger than $d = 0.01$, which violates the guarantees of $\mathcal{C}_c$. It is, however, still possible to circumvent the incompleteness of the refinement by strengthening the assumptions of $\mathcal{G}_{c1\parallel c2}$ to limit the size of the platoon to less than 10 vehicles, and by "propagating" this restriction to $\mathcal{G}_c$ via a mechanism of *assumption propagation*, as detailed below.

6.4.2.1 Assumptions Propagation.

When eliciting the top-level goals of a specification in a hierarchical way, we may discover additional assumptions, associated with lower-level goals in the hierarchy, which were not known *a priori*. This inconsistency between assumptions at different levels of the hierarchy may be a reason for incompleteness in the refinement. Let $\mathcal{G}_a$ and $\mathcal{G}_r$ be two goals, and $\mathcal{C}_a = (\psi_a, \phi_a)$ and $\mathcal{C}_r = (\psi_r, \phi_r)$ their respective contracts. Assumption propagation ensures that $\psi_a \rightarrow \psi_r$ is valid, by propagating the assumption formula from the lower-level contract to the upper-level contract and conjoining it with the assumptions of the higher-level contract so that behaviors that are not in ψ_r are removed from ψ_a and ψ_a is redefined as $(\psi_a \wedge \psi_r)$. After assumption propagation, the top-level guarantees will also be updated by bringing $\mathcal{C}_a$ again in saturated form, i.e., by setting the guarantees to $\phi_a \vee \overline{(\psi_a \wedge \psi_r)}$. In our example, the new assumptions for $\mathcal{C}_c$ after propagation become

$$\psi_c = \ 3 < f < 27 \ \wedge \ data \ \wedge \ n < 10,$$

which makes refinement complete and allows creating a refinement node in the CGT as in Figure 6.1.

6.4.3 Goal Manipulation via Conjunction

CoGoMo uses conjunction to generate a goal that refines multiple goals or *scenarios*, which are active under different assumptions, and may not be simultaneously satisfiable. In our example, the goal $\mathcal{G}_s$, 'Regulate Speed,' in Figure 6.1 can be decomposed into two sub-goals, $\mathcal{G}_1$ and $\mathcal{G}_2$, specifying how the speed of a vehicle s should be regulated according to the leader's speed s_l or according to the distance d_{front} to the front vehicle, respectively. Because $\mathcal{G}_1$ and $\mathcal{G}_2$ should both be satisfied, albeit in different situations, we can define a single goal $\mathcal{G}_s = \mathcal{G}_1 \ \wedge \ \mathcal{G}_2$ for the system.

If the assumptions of $\mathcal{G}_1$ and $\mathcal{G}_2$ are not mutually exclusive, the conjunction may require that potentially conflicting guarantees be satisfied simultaneously. For example, one of the contracts contributing to $\mathcal{G}_1$ may prescribe an *increase* of the follower's speed when there is an increase in the leader's speed, i.e., $\mathcal{C}_{11} = (s_t < s_l, \ \ s_{t+1} > s_t)$ where s_l, s_t and s_{t+1} represent the current speed of the leader and the speed of follower vehicle at times t and $t+1$, respectively. On the other hand, one of the contracts contributing to $\mathcal{G}_2$ may prescribe a *decrease* in the follower's speed when the distance to the vehicle in front is detected and it is less than a certain threshold, i.e., $\mathcal{C}'_{22} = (dist \wedge d_{\text{front}} < D_p, \ \ s_{t+1} < s_t)$, where *dist* evaluates to true if and only if the distance from the vehicle in front was detected correctly and $d_{\text{front}} < D_p$ indicates that the distance should be less than a constant (i.e., "the platooning distance"). The assumptions of $\mathcal{C}_{11}$ and $\mathcal{C}'_{22}$ can be satisfied simultaneously, possibly causing conflicts in the guarantees of the joint contract $(\psi_{11} \rightarrow (s_{t+1} > s_t)) \ \wedge \ (\psi'_{22} \rightarrow (s_{t+1} < s_t))$. CoGoMo prevents such conflicts via a *goal priority* mechanism.

6.4.3.1 Goal Priority.

A *goal priority* mechanism $\mathcal{P}(\mathcal{G}_1, \mathcal{G}_2)$ avoids such potential conflicts by making the assumptions mutually exclusive so that only one goal is effective under any environment. For instance, a priority mechanism may set $\mathcal{C}_2 = (\psi_2 \wedge \overline{\psi_1}, \phi_2)$, assuming that $\psi_2 \wedge \overline{\psi_1}$ is satisfiable, so that $\mathcal{C}_1$ is granted higher priority and dominates whenever both ψ_2 and ψ_1 hold. Because the assumptions of $\mathcal{C}_2$ become stronger, an assumption propagation step may also be needed to keep the refinement relations correct across the CGT. In our example, prioritizing $\mathcal{G}_1$ versus $\mathcal{G}_2$ solves the potential conflict. $\mathcal{G}_s$ can then be satisfied if the vehicle adjusts its speed according to the information provided by the distance sensor. When d_{front} is not available, as denoted by *dist*, the vehicle regulates its speed according to the speed of the leader of the platoon.

When composing a goal that was previously obtained by conjunction, e.g., when composing $\mathcal{G}_2$ with $\mathcal{G}_c$ in Figure 6.1, it may be useful to separately reason about the different scenarios involved in the composition. To do so, we use the fact that, given three contracts $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_c$, $[(\mathcal{C}_1 \wedge \mathcal{C}_2) \parallel \mathcal{C}_c] = [(\mathcal{C}_1 \parallel \mathcal{C}_c) \wedge (\mathcal{C}_2 \parallel \mathcal{C}_c)]$ holds, and we can separately identify the scenarios associated with $(\mathcal{C}_1 \parallel \mathcal{C}_c)$ and $(\mathcal{C}_2 \parallel \mathcal{C}_c)$ in the composition. In words, while composition is not distributive over conjunction in general [13], the distributive property

holds in the special case of three contracts as above. A proof of this result is in the appendix.

6.5 CGT Extension

Given a leaf node $\mathcal{G}'$ in a CGT and a library of goals $\mathcal{L}$, the extension problem consists in finding a set of goals $\mathcal{G}_1$, $\mathcal{G}_2$, ..., $\mathcal{G}_n$ in $\mathcal{L}$, that, once composed, refine $\mathcal{G}'$. The CGT is then extended by linking $\mathcal{G}'$ to a node $\mathcal{G}_s$ via refinement, and $\mathcal{G}_s$ to $\mathcal{G}_1$, $\mathcal{G}_2$,...,$\mathcal{G}_n$ via composition. Formally, we require:

$$\mathcal{C}_s = \mathcal{C}_1 \parallel \mathcal{C}_2 \parallel .... \parallel \mathcal{C}_n = (\psi_s, \phi_s),$$

$$\text{where} \quad \psi_s \wedge \psi' \text{ is satisfiable} \quad \text{and} \quad \phi_s \rightarrow \phi' \text{ is valid.}$$

When connecting $\mathcal{G}_s$ to $\mathcal{G}'$ via a refinement edge, CoGoMo also uses assumption propagation, as described in Section 6.4.2.1, to ensure that $\psi' \rightarrow \psi_s$ is valid.

Algorithm 2 proposes a procedure to automatically extend a CGT leaf node. The algorithm takes as inputs a goal node $\mathcal{G}'$ and a library of goals $\mathcal{L}$, and returns the set of goals to be composed as output. We assume that each goal in the library is labeled by a *cost* that is proportional to the number of clauses in the assumptions. The cost of a solution is the sum of the costs of all the selected goals. We first choose the lowest-cost selection of goals from $\mathcal{L}$ whose guarantees, once composed, imply the guarantees of $\mathcal{C}'$. Then, we choose the lowest-cost selection of goals from $\mathcal{L}$ whose guarantees, once composed, imply the assumptions of the goals selected in the previous iteration, and repeat this step until there are no more goals in the library or there are no assumptions that can be relaxed, i.e., discharged by the guarantees of another contract from the library. Concretely, given a input-goal $\mathcal{G}'$, where $\mathcal{C}' = (\psi', \phi')$, we look for all the combinations of goals in $\mathcal{L}$ such that their composition $\mathcal{C}_l = (\psi_l, \phi_l)$, has guarantees ϕ_l that imply either the guarantees of the input goal ϕ' (line 2) or its assumptions ψ' (line 7.1). We evaluate the cost of all the candidate compositions and select the candidate with the lowest cost (lines 3, 7.2). If multiple candidates have the same cost, we compose all the goals in each candidate set and select the composition that has the weakest assumptions.

Our cost metric favors the selection of goals with weaker assumptions (shorter assumption formulas), as they pose less constraints to the environment and support a larger number of contexts. On the other hand, a goal with a longer assumption formula tends to accept a smaller set of environments and require a more complex aggregation of goals from the library to discharge the assumptions. However, other cost functions are also possible. Searching for a composition of goals in the library that minimizes a cost function can be exponential in the size of the library. We circumvent the worst-case complexity by adopting a greedy strategy, which select the lowest-cost goal at each iteration, even if this does not necessarily lead to a globally optimal solution. As new goals are selected, they are aggregated via composition. Therefore, at each iteration, Algorithm 2 searches for the weakest-assumption contract whose guarantees discharge the assumptions of the composite contract obtained in the previous iteration.

As an example, we extend $\mathcal{G}'$ in Figure 6.1, which specifies the precision with which the distance from the vehicle in front is retrieved, when this

Algorithm 2: Goals Selection

Input: Library of goals $\mathcal{L} = \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_n\}$ with contracts $\{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_n\}$ respectively, input-goal $\mathcal{G}'$ with contract $\mathcal{C}' = (\psi', \phi')$

Output: Set of goals $\mathcal{L}_c \subseteq \mathcal{L}$ that composed would result in a goal $\mathcal{G}_l$ and contract $\mathcal{C}_l = (\psi_l, \phi_l)$ such that $\psi_l \wedge \psi'$ is satisfiable and $\phi_l \rightarrow \phi'$ is valid.

1 $\mathcal{L}_c = \emptyset$ is the set of goals to be returned

2 $R = \{(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_4), (\mathcal{G}_1, \mathcal{G}_5), (\mathcal{G}_2, \mathcal{G}_3), ...\}$ is a set of *candidate compositions*, where each element $R_i \subseteq \mathcal{L}$ and the composition of the goals contained in R_i results in a goal with a contract $\mathcal{C}_p = (\psi_p, \phi_p)$, such that

- $\psi_p \wedge \psi'$ is a satisfiable formula
- $\phi_p \rightarrow \phi'$ is a valid formula

3 $K = \{\mathcal{G}_i, \mathcal{G}_j, .., \mathcal{G}_m\} =$ **optimal_selection**(R)

4 $\mathcal{L}_c \leftarrow K$ adds the selected goals to $\mathcal{L}_c$

5 $\mathcal{S} \leftarrow K$ where $\mathcal{S}$ is the set of goals whose assumptions need to be searched in the library

6 **while** $S \neq \emptyset$ **do**

7 **for** *goal* $\mathcal{G}_s$ ***in*** $\mathcal{S}$ *where* $\mathcal{C}_s = (\psi_s, \phi_s)$ **do**

7.1 $Q = \{(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3), (\mathcal{G}_2, \mathcal{G}_5), (\mathcal{G}_1, \mathcal{G}_3), ...\}$ is a set of *candidate compositions*, where each element $Q_i \subseteq \mathcal{L}$ and the composition of the goals in Q_i has a contract $\mathcal{C}_q = (\psi_q, \phi_q)$ such that:

- $\psi_q \wedge \psi' \wedge \psi_s$ is a satisfiable formula
- $\phi_q \rightarrow \psi_s$ is a valid formula

7.2 $H = \{\mathcal{G}_i, \mathcal{G}_j, .., \mathcal{G}_m\} =$ **optimal_selection**(Q)

7.3 $\mathcal{L}_c \leftarrow \mathcal{L}_c \cup H, \quad \mathcal{S} \leftarrow S \cup H$

7.4 $\mathcal{S} \leftarrow \mathcal{S} - \{\mathcal{G}_s\}$ removes the $\mathcal{G}_s$ from $\mathcal{S}$

8 **return** $\mathcal{L}_c$

information is available. The associated contract $\mathcal{C}' = (--, \ dist \ \wedge \ d_{\text{front}} > 0 \ \wedge \ |d_{\text{front}} - d_{\text{real}}| < \delta)$ guarantees that the information on the distance *dist* is available and that the perceived distance with the vehicle in front d_{front} is positive and has a precision δ in all contexts, where δ is a constant. We then use a library of goals specifying GPS modules, accelerometers, several kinds of radars with different levels of accuracy, and communication components. The extension algorithm returns 6 goals whose composition $\mathcal{G}_s$ is linked via refinement to $\mathcal{G}'$ in Figure 6.3.

The left-hand side of Figure 6.3 shows the contracts formalizing the new goal $\mathcal{G}_s$ and their interconnection structure. Each edge between contracts is labeled with a proposition that represents the logic predicate forming the assumptions or the guarantees of a contract (or both in case the guarantees of one contract imply some of the propositions in the assumptions of another contract). For example, the contract $\mathcal{C}_3$ in Figure 6.3, which specifies a Kalman Filter component, has assumptions $a \wedge c_{ego}$ and guarantees p. The composition

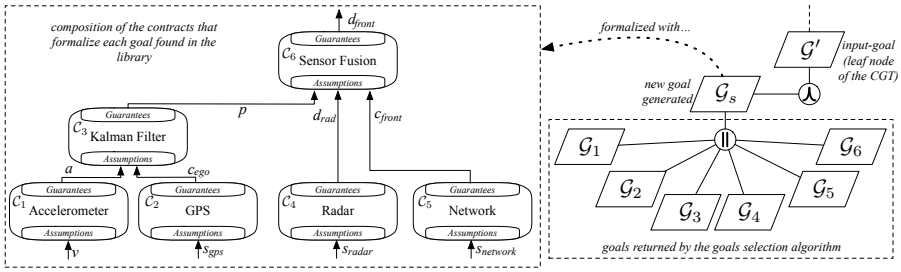


Figure 6.3: Example of CGT extension via a library of goals. The left-hand side shows the composition of contracts used to formalize $\mathcal{G}_s$ and extend the CGT on the right-hand side.

of $\mathcal{C}_3$ with $\mathcal{C}_1 = (v, a)$ and $\mathcal{C}_2 = (s_{gps}, c_{ego})$ results in a contract, where a and c_{ego} are no longer present in the assumptions, since they are already supported by the guarantees. The net result is a simpler assumption formula. By composing all the goals retrieved from the library we obtain a new goal $\mathcal{G}_s$ and associated contract $\mathcal{C}_s = (\psi_s, \phi_s)$, where

$$\begin{aligned}\psi_s &= v \wedge s_{gps} \wedge s_{radar} \wedge s_{network} \wedge a \wedge c_{ego} \wedge p \wedge d_{rad} \wedge c_{front} \\ &\quad \vee \overline{(a \wedge c_{ego} \wedge p \wedge d_{rad} \wedge c_{front} \wedge d_{front})} \\ \phi_s &= a \wedge c_{ego} \wedge p \wedge d_{rad} \wedge c_{front} \wedge d_{front}.\end{aligned}$$

As in the previous example, the assumption formula reduces to $\psi_s = (v \wedge s_{gps} \wedge s_{radar} \wedge s_{network} \vee \overline{d_{front}})$. We observe that ϕ_s refines the guarantees of $\mathcal{C}'$ because $d_{front} = dist \wedge d_{\text{front}} > 0 \wedge |d_{\text{front}} - d_{\text{real}}| < \epsilon$ where ϵ is a constant and $\epsilon \leq \delta$. Finally, to preserve the completeness of the refinement, CoGoMo propagates the assumptions ψ_s to $\mathcal{C}'$ and then to the parent nodes of $\mathcal{G}'$ recursively, by following the edges of the CGT up to the root.

6.5.0.1 Numerical Validation.

Algorithm 2 is sound and complete. The soundness is provided by the SMT solver, which checks the validity and satisfiability of the formulas. The completeness is given by the fact that the algorithm searches over the entire goal library. Because of the greedy procedure, the computation time scales linearly with the number of goals in the library. We performed numerical evaluation of synthetically generated libraries of different sizes populated by randomly generated goals. Goals are captured by simple propositional logic contracts, whose assumptions and guarantees are conjunctions of Boolean propositions. We use the length of these formulas, i.e., to quantify the complexity of each contract. A configuration is defined by the number of goals in the library and the complexity of the contracts. We evaluated the algorithm on up to 1000 library goals and up to 24 logical propositions in each contract for a total of 50 different configurations. For each configuration, we ran Algorithm 2 with 100 different input goals. Table 6.1 shows the average execution time for each configuration normalized by the number of goals returned by the algorithm. Figure 6.4 shows the execution times for 3 configurations, which scale linearly

	100	200	300	400	500	600	700	800	900	1000
4	0.49	0.94	1.32	1.78	2.14	2.53	2.99	3.45	3.82	4.09
8	1.42	2.71	3.89	5.20	6.49	7.73	9.09	10.41	11.82	13.10
16	4.84	10.74	16.04	18.97	23.27	29.05	35.16	36.85	41.30	46.61
20	6.33	12.62	18.40	24.58	32.70	39.83	45.45	56.09	63.83	70.33
24	8.79	17.14	25.59	34.07	49.45	54.98	64.34	71.91	84.10	94.13

Table 6.1: Average execution times (sec) of 100 runs for different configurations of library size (number of goals in the columns) and contract complexity (rows).

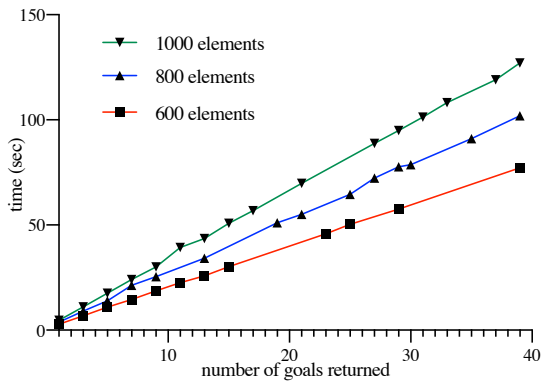


Figure 6.4: Execution times as a function of the number of returned goals for 3 simulation configurations, with libraries of 600, 800, and 1000 elements, respectively, and contract complexity equal to 4.

with the number of returned goals and the size of the library.[1]

6.6 Related Work

In the context of contract-based verification, tools like OCRA [191] and AGREE [192] use contracts to model system components and their aggregations and formally prove the correctness of contract refinements [193] by using model checkers [177]. Related to system engineering, CONDEnSe [194] propose a methodology and a tool that leverages the algebra of contracts to integrate artifacts developed in mechatronic systems. More oriented toward requirement engineering, the CHASE [179] framework combines a front-end formal specification language based on patterns with rigorous, contract-based verification and synthesis. It uses a declarative style to define the top-level requirements that are then translated into temporal logic, verified for consistency and, when possible, synthesized into a reactive model. CoGoMo's hierarchical and incremental approach to refinement of goal models is complementary and can be naturally incorporated into CHASE.

In the context of goal-oriented requirement engineering, significant work has addressed completeness and conflict detection using formal methods for goal models based on KAOS [181, 182, 195]. While these approaches mostly

[1] Complete results: `http://bit.ly/3s5XD0L`.

focus on algorithms that operate on a fixed set of requirements and environment expectations, CoGoMo proposes a step-wise approach where refinement checking and conflict analysis are performed contextually in an incremental way as the goal tree is built. Frameworks like COVER [196] use TROPOS as a goal modeling framework, Modal Transition Systems (MTS) to model the system design, and Fluent Linear Temporal Logic (FLTL) as a specification language for functional requirements. COVER checks the satisfaction of all the requirements by verifying the properties on the system model. Requirement verification using formal methods is common to the goal-oriented approaches above; however, to the best of our knowledge, CoGoMo is the first effort toward formalizing goal models using contracts, thus enhancing modularity and reuse in goal models.

Compositional synthesis of reactive systems, i.e., finding generic aggregations of reactive components such that their composition realizes a given specification, is an undecidable problem [197, 198]. The problem becomes decidable by imposing a bound on the total number of component instances that can be used, but remains difficult due to its combinatorial nature [199]. Our approach relates to the one by Iannopollo et al. [176,199], proposing scalable algorithms for compositional synthesis and refinement checking of temporal logic specifications out of contract libraries. Our goal selection algorithm is, however, different, as it uses a cost function based on the complexity of a specification, and a greedy procedure that favors more compact and generic specifications (i.e., contracts with weakest assumptions) to refine the goal tree, while keeping the problem tractable.

6.7 Conclusions

We presented CoGoMo, a framework that guides the designer in building goal models by leveraging a contract-based formalism. CoGoMo leverages contract operations and relations to check goal consistency, completeness, and support the incremental and hierarchical refinement of goals from a library of goals. An example motivated by vehicle platooning shows its effectiveness for incrementally constructing contract-based goal trees in a modular way, with formal guarantees of correctness. Numerical results also illustrate the scalability of the proposed greedy heuristic to further extend a goal tree out of a library of goals. As future work, we plan to extend the expressiveness of CoGoMo by i) supporting contracts expressed in temporal logic and ii) supporting OR-refinements between goals by allowing optional refinement relations between multiple candidate contracts. Furthermore, we plan to improve the tool and incorporate its features into CROME [200], our recent framework for formalizing, analyzing, and synthesizing robotic missions.

Acknowledgments

The authors wish to acknowledge Alberto Sangiovanni-Vincentelli, Antonio Iannopollo, and igo ncer Romeo for helpful discussions. This work was supported in part by the Wallenberg Al Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation. In

addition, the authors gratefully acknowledge the support by the US National Science Foundation (NSF) under Awards 1846524 and 1839842, the US Defense Advanced Projects Agency (DARPA) under Award HR00112010003, the US Office of Naval Research (ONR) under Award N00014-20-1-2258, and Raytheon Technologies Corporation. The views, opinions, or findings contained in this article should not be interpreted as representing the official views or policies, either expressed or implied, by the US Government. This content is approved for public release; distribution is unlimited.

Appendix

Distribution of Composition over Conjunction Given contracts $\mathcal{C}_1 = (\psi_1, \phi_1)$, $\mathcal{C}_2 = (\psi_2, \phi_2)$, and $\mathcal{C}_3 = (\psi_3, \phi_3)$, we show that

$$(\mathcal{C}_1 \wedge \mathcal{C}_2) \parallel \mathcal{C}_3 = (\mathcal{C}_1 \parallel \mathcal{C}_3) \wedge (\mathcal{C}_2 \parallel \mathcal{C}_3). \tag{6.2}$$

Proof. Let (L_A, L_G) and (R_A, R_G) be the contracts on the left and right side of (6.2), respectively. We prove that $L_A = R_A$ and $L_G = R_G$. Both the composition and conjunction operations requires the conjunction of the guarantees, hence we obtain $L_G = R_G = \phi_1 \wedge \phi_2 \wedge \phi_3$. The assumptions of the contract on the left side can be computed as

$$L_A = (\psi_1 \vee \psi_2) \wedge \psi_3 \vee \overline{(\phi_1 \wedge \phi_2 \wedge \phi_3)} = (\psi_1 \wedge \psi_3) \vee (\psi_2 \wedge \psi_3) \vee \overline{\phi}_1 \vee \overline{\phi}_2 \vee \overline{\phi}_3.$$

On the right side, we obtain

$$\mathcal{C}_1 \parallel \mathcal{C}_3 = \left((\psi_1 \wedge \psi_3) \vee \overline{(\phi_1 \wedge \phi_3)}, \phi_1 \wedge \phi_3\right) \text{ and}$$
$$\mathcal{C}_2 \parallel \mathcal{C}_3 = \left((\psi_2 \wedge \psi_3) \vee \overline{(\phi_2 \wedge \phi_3)}, \phi_2 \wedge \phi_3\right),$$

which leads to

$$\begin{aligned} R_A &= (\psi_1 \wedge \psi_3) \vee \overline{(\phi_1 \wedge \phi_3)} \vee (\psi_2 \wedge \psi_3) \vee \overline{(\phi_2 \wedge \phi_3)} \\ &= (\psi_1 \wedge \psi_3) \vee (\psi_2 \wedge \psi_3) \vee \overline{\phi}_1 \vee \overline{\phi}_2 \vee \overline{\phi}_3. \end{aligned}$$

Finally, we also obtain $L_A = R_A$, which concludes our proof (6.2). □

Chapter 7

Paper F

A Framework for Specifying and Realizing Correct–by–Construction Contextual Robotic Missions using Contracts

P. Mallozzi, G. Schneider, N. Piterman, P. Nuzzo, and P. Pelliccione

in submission.

Abstract

Automatically synthesizing robotic mission controllers that are proven to be correct from a formal specification is a hard problem that can be intractable if the mission specification is too complicated. Decomposing a mission specification in various sub-missions can make the synthesis of independent parts of the mission easy to achieve; however, we lose the guarantees on the overall mission correctness.

In this paper, we address the problem of dynamic switching of controllers of a distributed robotic mission while providing guarantees of correctness on the overall mission specification under certain conditions. Our framework allows us to model the overall mission in terms of mission *scenarios* that are enabled under certain mission *contexts*. We represent the mission in terms of a hierarchy of reusable goals, where each goal is rooted in formal contract-based representations. Overall our framework, named CROME, allows us to: *i) decompose* the mission in several independent clusters, each containing mission scenarios acting under a certain context, *ii) automatically refine* the mission using pre-defined and reusable goals, *iii) synthesize* controllers for each cluster and at different abstraction levels, and *iv) dynamically switch* from one controller to another at run-time while formally proving the correctness of the overall mission.

7.1 Introduction

Service robots are robots that perform useful tasks for humans or equipment excluding industrial automation applications [164]. They are increasingly used in various domains, such as healthcare, logistics, telepresence, infrastructure maintenance, education, domestic tasks, and entertainment. The development of service robots leads to increasing software engineering challenges since they require sound software development practices with high levels of robustness and autonomy to operate in highly heterogeneous environments, often shared with humans [168]. Solving these challenges will promote a "shift towards well-defined engineering approaches able to stimulate component supply-chains and significantly impact the robotics marketplace" [163].

One of the software-engineering challenges concerns the mission specification [168], which describes the high-level tasks the robotic software must accomplish [201]. By following the terminology in [48], we distinguish between *mission requirements*, which describe the robotic mission in natural language, and *mission specifications*, which formulate mission requirements unambiguously and precisely. Writing a mission specification taking into account the variability of conditions of application scenarios in real operational environments is a complex and tedious task [168, 202]. In fact, robots are required to operate in the real world, where (i) the variability of the environment is high, (ii) there can be various error cases, and (iii) robots might even change the environment itself via actions. Covering all these cases might be intractable in certain scenarios. Moreover, automatically synthesizing robotic mission controllers that are proven to be correct from a formal specification is a hard problem that can be intractable if the mission specification is too complicated. Decomposing a mission requirement in various sub-missions can solve both

problems, by making the specification easier and also computationally more tractable. The downside of the mission decomposition is that we might lose the guarantees on the overall mission correctness.

Various approaches have been proposed in recent years to specify missions, spanning from approaches making use of logics [32, 33, 35–43], to approaches making use of state charts [203–205], Petri Nets [206, 207] or domain specific languages [63, 208–213]. Moreover, the work in [48] presents a catalog of 22 mission specification patterns for mobile robots. A pattern provides a solution to recurrent specification problems, enabling the description of a mission requirement in natural language (English) and offering a translation to a mission specification in temporal logic. However, to the best of our knowledge, the problem of specifying a mission in terms of sub-missions and at the same time guaranteeing the correctness of the overall mission specification under certain conditions is still an open problem.

In CROME [200] we have addressed the problem of capturing robotic mission requirements, analyze them and ultimately synthesize them in multiple controllers implementing different parts of the mission. We have introduced the concept of *contexts*, i.e. different environmental conditions that can occur during the mission, and mission *scenarios*, i.e. sub-missions that are active only under a certain context. The designer has to provide all the domain properties for each goal, that is *rules* of the mission domain such as mutually exclusive locations. The framework leverages a formal model, termed Contract-based Goal Graph (CGG), which enables organizing the requirements in a modular way with rigorous compositional semantics.

However, in our previous work, we did not allow any *context-switch* to happen during a mission, i.e. the robot can not switch from one controller realizing a mission scenario to another controller which is supported under another context. Furthermore writing explicitly all the domain rules for each goal can be greatly automatized. Finally, the CGG did not capture all possible meaningful connections among goals that we could have.

In this paper we extend CROME to overcome these limitations. The result is a framework that enables the specification of robust missions in heterogeneous environments characterized by high variability. In the context of this paper we consider that a robotic mission is formed by several tasks, where each task is defined as a robotic specification that has to hold in a specific context.

7.1.1 Main contributions

The main contributions of this paper to the specification of robotic missions are:

— *Automatic inference of domain rules.* We introduce all the elements forming CROME and present how we can automatically infer the domain rules of the mission. Our new infrastructure allows CROME to automatically derive the mission constraints related to the different goals of the mission without having the designer manually explicit them for every goal.

— *Enhanced Formal Representation of the Mission Specification.* We have enhanced the CGG, i.e. the formal model presented in [200] representing

a graph of goals. Each node of the graph can now be linked to other nodes via different types of connection (e.g. a node can be linked both by refinement and by compositions), enhancing the modularity of the CGG.

— *Orchestrating mission controllers to support context switching throughout the mission.* The most significant contribution is the possibility for the robot controller to perform a mission while the context is being changed. Each context is related to one *specification controller*. We have introduced a new formal methodology to automatically switch from one specification controller to another while providing guarantees on the satisfaction of the overall missions.

CROME leverages on *i)* assume-guarantee contracts [13, 14] to formalize mission goals, *ii)* robotic patters to generate mission specifications, *iii)* reactive synthesis to produce mission controllers and *iv)* the concept of *context* to characterize different parts of the mission.

7.1.2 Running Example and Mission Requirements

Consider the topology depicted in Figure 7.1. We have five locations/regions, i.e. $\Omega_R = \{r_1, r_2, r_3, r_4, r_5\}$, two contexts $\{day, night\} \in \Omega_X$, one sensor indicating the presence of a person, i.e. $\Omega_S = \{\text{person}\}$ and two actions, i.e. $\Omega_A = \{\text{greet, register}\}$. The mission consists in a robot that has to patrol different locations according to the context. During the *day* it has to patrol locations r_1 and r_2 in order, starting from r_1. During the night it has to patrol locations r_3 and r_4 in order, starting from r_3. Whenever there is a context change, the robot has to resume the patrolling task from the last visited location. Furthermore, whenever the robot senses a person in any location it activates the greeting action immediately. Finally, only during the day, whenever a person is detected, the robot must register them promptly.

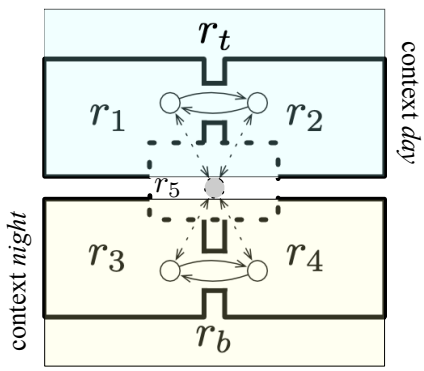


Figure 7.1: Running example consisting of five regions and two contexts.

7.1.3 Roadmap of the paper

In Section 7.2, we present the necessary background on assume-guarantee contracts, robotic patterns, reactive synthesis and context and we introduce related

works in mission specification. In Section 7.3, we present the infrastructure offered by CROME to model mission specifications from mission requirements. In Section 7.4, we present the problems of generating contextual controllers from the mission specification, and, in Section 7.5, we show how CROME addresses the problems and generates specification controllers which realize the mission. Section 7.6 shows the orchestration in the running example. Section 7.7 shows the relationship between mission satisfaction and CGG satisfaction. Section 7.8 presents the proof of concept tool and describes the evaluation we performed. Finally, Section 7.9 concludes with final remarks and future research directions.

7.2 Background and Related works

In this section, we provide some background on the basic building blocks of CROME, namely, contracts, linear temporal logic, reactive synthesis and specification patterns, and we provide an overview of related works in robotic mission specification.

7.2.1 Assume-Guarantee Contracts

Contract-based design [13, 14] has emerged over the years as a design paradigm capable of providing formal support for building complex systems in a modular way, by enabling compositional reasoning, stepwise refinement, and reuse of pre-designed components.

A *contract* $\mathcal{C}$ is a triple (V, A, G) where V is a set of system *variables* (including, e.g., input and output variables or ports), and A and G – the assumptions and guarantees – are sets of behaviors over V. For simplicity, whenever possible, we drop V from the definition and refer to contracts as pairs of assumptions and guarantees, i.e., $\mathcal{C} = (A, G)$. A expresses the behaviors that are expected from the environment, while G expresses the behaviors that an implementation promises under the environment assumptions. In this paper, we express assumptions and guarantees as sets of behaviors satisfying a logic formula; we then use the formula itself to denote them, with a slight abuse of notation, whenever there is no confusion. An environment E satisfies a contract $\mathcal{C}$ whenever E and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of E are included in the assumptions of $\mathcal{C}$, i.e., when $|E| \subseteq A$, where $|E|$ is the set of behaviors of E. An implementation M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables and all the behaviors of M are included in the guarantees of $\mathcal{C}$ when considered in the context of the assumptions A, i.e., when $|M| \cap A \subseteq G$.

A contract $\mathcal{C} = (A, G)$ can be placed in saturated form by re-defining its guarantees as $G_{sat} = G \cup \overline{A}$, where $\overline{A}$ denotes the complement of A. A contract and its saturated form are semantically equivalent, i.e., they have the same set of environments and implementations. Therefore, in the rest of the paper, we assume that all the contracts are expressed in saturated form. A contract $\mathcal{C}$ is *compatible* if there exists an environment for it, i.e., if and only if $A \neq \emptyset$. Similarly, a saturated contract $\mathcal{C}$ is *consistent* if and only if there exists an implementation satisfying it, i.e., if and only if $G \neq \emptyset$. We say that a contract is *well-formed* if and only if it is compatible and consistent. We detail below the contract operations and relations used in this paper.

7.2.1.1 Contract Refinement

Refinement establishes a pre-order between contracts, which formalizes the notion of replacement. Let $\mathcal{C} = (A, G)$ and $\mathcal{C}' = (A', G')$ be two contracts. $\mathcal{C}$ refines $\mathcal{C}'$, denoted by $\mathcal{C} \preceq \mathcal{C}'$, if and only if all the assumptions of $\mathcal{C}'$ are contained in the assumptions of $\mathcal{C}$ and all the guarantees of $\mathcal{C}$ are included in the guarantees of $\mathcal{C}'$, that is, if and only if $A \supseteq A'$ and $G \subseteq G'$. Refinement entails relaxing the assumptions and strengthening the guarantees. When $\mathcal{C} \preceq \mathcal{C}'$, we also say that $\mathcal{C}'$ is an *abstraction* of $\mathcal{C}$ and can be replaced by $\mathcal{C}$ in the design.

7.2.1.2 Contract Composition

Contracts associated with distinct implementations can be combined via the composition operation ($\|$) to specify the composition between the corresponding implementations. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. The composition $\mathcal{C} = (A, G) = \mathcal{C}_1 \parallel \mathcal{C}_2$ can be computed as follows:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}, \tag{7.1}$$

$$G = G_1 \cap G_2. \tag{7.2}$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, hence the operation of intersection in (1.2). An environment for $\mathcal{C}$ should also satisfy all the assumptions, motivating the conjunction of A_1 and A_2 in (1.1). However, part of the assumptions in $\mathcal{C}_1$ may be already supported by $\mathcal{C}_2$ and *vice versa*. This allows relaxing $A_1 \cap A_2$ with the complement of the guarantees of $\mathcal{C}$ [13].

7.2.1.3 Contract Conjunction

Different contracts on a single implementation can be combined using the conjunction operation ($\wedge$). Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be two contracts. We can compute their conjunction by taking the greatest lower bound of $\mathcal{C}_1$ and $\mathcal{C}_2$ with respect to the renement relation. Intuitively, the conjunction $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ is the weakest (most general) contract that refines both $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}$ can be computed by taking the intersection of the guarantees and the union of the assumptions, that is:

$$\mathcal{C} = (A_1 \cup A_2, G_1 \cap G_2).$$

Intuitively, an implementation satisfying $\mathcal{C}$ must satisfy the guarantees of both $\mathcal{C}_1$ and $\mathcal{C}_2$, while being able to operate in either of the environments of $\mathcal{C}_1$ or $\mathcal{C}_2$.

7.2.2 Linear Temporal Logic

Given a set of atomic propositions AP (i.e., Boolean statements over system variables) and the state s of a system (i.e., a specific valuation of the system variables), we say that s *satisfies* p, written $s \models p$, with $p \in AP$, if p is *true* at state s. We can construct LTL formulas over AP according to the following recursive grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\ \varphi \mid \varphi_1\ \mathbf{U}\ \varphi_2$$

where φ, φ_1, and φ_2 are LTL formulas. From the negation ($\neg$) and disjunction ($\vee$) of the formula we can define the conjunction ($\wedge$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). Boolean constants *true* and *false* are defined as $true = \varphi \vee \neg\varphi$ and $false = \neg true$. The temporal operator **X** stands for *next* and **U** for *until*. Other temporal operators such as *globally* (**G**) and *eventually* (**F**) can be derived as follows: $\mathbf{F}\ \varphi = true\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \varphi = \neg(\mathbf{F}(\neg\varphi))$. We refer to the literature [16] for the formal semantics of LTL.

7.2.3 Reactive Synthesis

Given an LTL specification φ over a set of atomic proposition partitioned into inputs and outputs, i.e. $AP = \mathcal{I} \cup \mathcal{O}$, the *synthesis problem* determines whether there exists a finite-state machine $\mathcal{M} = (S, \mathcal{I}, \mathcal{O}, s_0, \delta)$ that satisfies φ. If such machine exists it computes it and we say that $\mathcal{M}$ *realizes* the formula φ.

A finite state machine is a tuple $\mathcal{M} = (S, \mathcal{I}, \mathcal{O}, s_0, \delta)$ where S is the set of states, $s_0 \in S$ is the initial state, and $\delta : S \times 2^{\mathcal{I}} \rightarrow S \times 2^{\mathcal{O}}$ is the transition function. We say that some word $w = (w_0^{\mathcal{I}}, w_0^{\mathcal{O}})(w_1^{\mathcal{I}}, w_1^{\mathcal{O}})(w_2^{\mathcal{I}}, w_2^{\mathcal{O}})... \in (2^{\mathcal{I}} \times 2^{\mathcal{O}})^{\omega}$ is a *trace* of $\mathcal{M}$ if there exists a run $\tau = \tau_0\tau_1\tau_2... \in S^{\omega}$ such that $\tau_0 = s_0$ and for every $i \in \mathbb{N}$ we have that $(\tau_{i+1}, w_i^{\mathcal{O}}) = \delta(\tau_i, w_i^{\mathcal{I}})$. A machine $\mathcal{M}$ satisfies a formula φ if all its runs satisfy φ.

Reactive synthesis has been applied in robotics for synthesizing controllers. One example is the language Spectra [214], which offers high-level constructs such as patterns and monitors to specify assumptions and guarantees for a reactive system. The work in [215] presents a compositional approach to perform reactive synthesis for finite-horizon tasks represented using formulas of linear temporal logic on finite traces (LTLf). The work in [216] presents algorithms to synthesize controllers for a swarm robotic system. The work in [217] identifies some software engineering challenges in applying reactive synthesis to robotics. The challenges include the synthesis algorithms themselves, adequate development processes and tools, declarative specifications writing, and abstraction of data and time. In this work, we present CROME, which synthesizes a set of controllers, one for each context and satisfying the contextual mission, and orchestrates the controllers so to satisfy the overall contextual mission.

7.2.4 Mission specification and Robotic Patterns

When dealing with mission specification it is important to distinguish between specifications that require the involvement of end-users and those that, instead, do not require the user involvement [168]. If the end-user is directly involved, usability and simplicity of mission specification become important requirements.

Various approaches have been proposed in the last years to specify missions. Some of them make use of logics [33,35–40], and indeed they can be not the best solution in the case of end-users directly involved in the mission specification. Other approaches make use of state charts [203–205] or Petri Nets [206,207], and also in this case they require special knowledge and skills in their end-user. To make the mission specification more accessible, various researchers investigated and proposed domain specific languages for mission specification [63, 208–213]. The work in [218] surveys domain-specific languages (DSLs) for robot mission specification, classify them in internal or external DSLs and gives an overview

of their tooling support.

Moreover, many commercial service robots are equipped with environments and domain-specific languages tailored for end-users. The work in [219] surveys 30 of these mission specification environments for mobile robots. The work explores the design space of these environments and underlying languages and classifies them by the kinds of syntax they offer: block-,flowchart-, graph-, text- or map-based syntaxes.

However, mission specification is still considered to be a challenge and often the most difficult aspect concerns not much specifying what the robot should do, but, instead, the specification of "*exceptional behaviors to specify how robots ought to cope with the variability of conditions of application scenarios in real environments in which robots are required to operate*" [168].

A way to deal with this challenge is to decompose the mission specification is sub-mission specification, each activated in special contexts and conditions. The work in [220] proposes the automatic computation of a controller to handle the transition from one specification to another, so from one controller to another. The synthesis of controllers is done at runtime; this brings flexibility but might creates problems if there is the need to instantaneously switch from one controller to another. The approach guarantees the preservation of "transition properties" when switching from one controller to another. The work in [221] proposes a hierarchy of discrete event controllers that support graceful degradation when the assumptions of a higher level are broken, and progressive enhancement when those assumptions are satisfied or restored. The approach allows the instantaneous switch and offers guarantees regarding the behaviors across controller switches. The main drawback of this approach concerns the fact that layers are predefined and must be organized into a strict logical implication order. Despite these precious contributions to the state-of-the-art in the field, to the best of our knowledge the problem of specifying a mission in terms of sub-missions and at the same time guaranteeing the correctness of the overall mission specification under certain conditions is still an open problem.

Robotic patterns have been proposed as a solution to recurrent mission specification problems based on the analysis of mission requirements in the robotic literature [48]. The PsALM tool [172] enables the specification of robotic missions using mission specification patterns.

CROME supports 22 patterns [48], capturing robot movements and actions performed as a robot move in the environment, organized into three groups: *core movement* patterns, *triggers*, and *avoidance* patterns.

For example, let us assume that the mission requirement is: 'A robot must patrol a set of locations in a certain strict order.' The designer can formulate this requirement by using the *Strict Ordered Patroling* pattern[1], instantiated for the required set of locations. Let l_1, l_2, and l_3 be the atomic propositions of type *location* that the robot must visit in the given order. The mission requirement can then be reformulated as 'Given the locations l_1, l_2, and l_3, the robot should visit all the locations indefinitely and following a strict order,'

[1] `http://roboticpatterns.com/pattern/strictorderedpatrolling/`

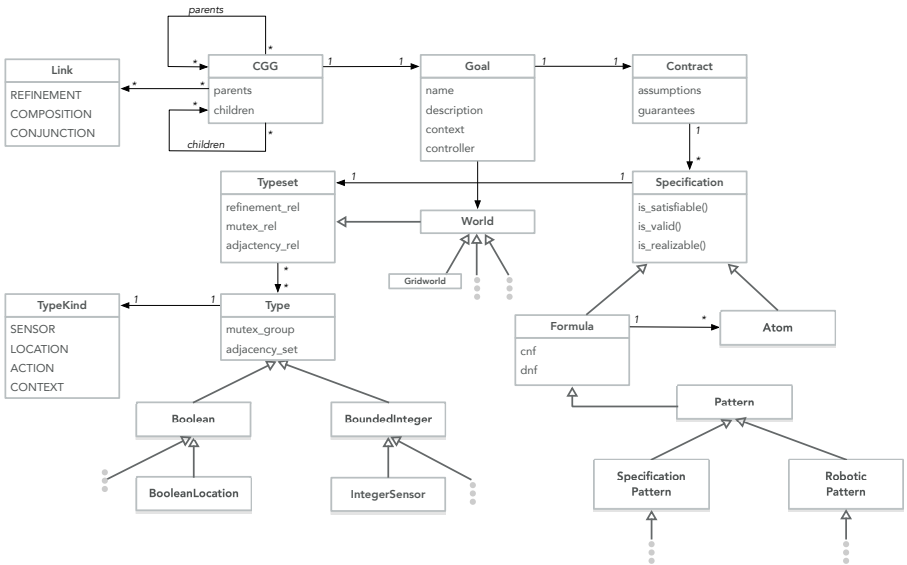


Figure 7.2: Main classes composing the CROME framework

leading to the following LTL formulation:

$$\begin{aligned} &\mathbf{G}(\mathbf{F}(l_1 \wedge \mathbf{F}(l_2 \wedge \mathbf{F}(l_3)))) \wedge (\overline{l_2}\ \mathbf{U}\ l_1) \wedge (\overline{l_3}\ \mathbf{U}\ l_2) \\ \wedge\ &\mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_1)) \wedge \mathbf{G}(l_3 \rightarrow \mathbf{X}(\overline{l_3}\ \mathbf{U}\ l_2)) \\ \wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_3)) \\ \wedge\ &\mathbf{G}(l_1 \rightarrow \mathbf{X}(\overline{l_1}\ \mathbf{U}\ l_2)) \wedge \mathbf{G}(l_2 \rightarrow \mathbf{X}(\overline{l_2}\ \mathbf{U}\ l_3)). \end{aligned} \tag{7.3}$$

As shown in this example, a robotic pattern can significantly facilitate the difficult and error-prone task of mission specification.

7.2.5 Contexts and mission-related contexts

Many characterizations of the 'context' of an application have been provided, often informally, in the literature. In context-aware ubiquitous computing [173], the context of an application may include information like location, identities of nearby people and objects, time of the day, season, or temperature. More generally, Dey and Abowd [174] define context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves." In the robotic domain, Bloisi et al. [175] define mission-related contexts as "choices that are useful in robotic scenarios for adapting the robot behaviors to the different situations." CROME builds on these characterizations and adapts them to robotic missions by formalizing a context as a property associated with a goal and expressed by a logic formula.

7.3 From Mission Requirements to Mission Specifications

The CROME framework allows designers to express mission requirements in a structured way using robotic patterns to specific mission requirements. In this section we describe the major elements characterizing the infrastructure of CROME and how they are related to each other. We show how the designer can use these elements to model mission requirements and produce mission specifications using the running example. Figure 7.2 shows the main classes composing the CROME framework. These main classes are described in the following.

Type A *Type* represents a variable in a domain. The basic types supported are *Boolean* and *BoundedInteger*. By extending the basic types, CROME builds more refined ones such as *BooleanLocation* and *IntegerSensor* representing variables indicating a physical location and variables indicating the value of a sensor withing a certain range, respectively. Each type has a *TypeKind* that can one of the following: *sensor*, *location*, *action* or *context*.

Depending on the kind of the type, we might have two additional properties for types extending the basic *Boolean* type: `mutex_group` and `adjacency_set`. For example, a type representing a physical location (e.g. *BooleanLocation*) might have a `mutex_group` indicating a group such that all the types belonging to the same group are considered *mutually exclusive*, i.e. its value cannot be assigned to *true* to more that one type in the group in the same time-step. The `adjacency_set` contains a set of types which are *adjacent*, i.e. that can become true in the *next* time-step. For example, in case of types representing locations the `adjacency_set` indicates all the regions adjacent to the location under consideration.

Typeset A *Typeset* is a set of *Type* elements and the relationships among them. The attributes `refinement_rel`,
`mutex_rel`, and `adjacency_rel` in each Typeset keeps track of the *refinement*, *mutually exclusion*, and *adjacency* relationship among the elements in the typeset, respectively. The refinement relationships are derived directly from the hierarchical structure of the types, i.e. the designer can define types which *extend* other types, creating a refinement relationship. Adjacency and mutually exclusion relationships are derived from the corresponding properties expressed in the types of the typeset.

World The *World* is a *Typeset* that exists independently from a certain specification. For example, the environment in our running example depicted in Figure 7.1 can be formulated as a *World* containing

— 2 *BooleanLocation* regions r_t and r_b, which are mutually exclusive in $group_1$.

— 5 *BooleanLocation* r_1, r_2, r_3, r_4, and r_5, which are mutually exclusive in $group_2$, where r_1 and r_2 *refine* (i.e. *extend*) r_t and r_3 and r_4 refine r_b.

We also have that r_1 is adjacent to r_2 and r_5, r_2 is adjacent to r_1 and r_5, and so on.

— 2 *ContextBooleanTime* types *day* and *night*, which extend the *Boolean* type and represent two contexts, e.g. mutually exclusive in $group_3$.

Specification A specification represents an LTL formula, which in CROME can be constructed with a pattern, i.e. a specification pattern or a robotic pattern. A pattern can either be an *Atom*, i.e. a single LTL expression that cannot be broken down in smaller chunks, or a *Formula*, i.e. a set of atoms connected via conjunction or disjunction. Each specification is defined over a Typeset. For example, let us have a Typeset containing two elements: *BooleanSensor* type s and the *BooleanAction* type a, both extending the basic Boolean Type. The robotic pattern *instant reaction* applied to the atomic propositions s and a is an *Atom* representing the LTL formula:

$$\mathbf{G}(s \rightarrow a) \tag{7.4}$$

Instead, the robotic pattern *visit* applied to the atomic propositions r_1 and r_2 (defined over *BooleanLocation* types) is a *Formula* containing the conjunction of two Atom elements and represented by the LTL formula:

$$\mathbf{F}(r_1) \wedge \mathbf{F}(r_2) \tag{7.5}$$

We will use the same symbol to indicate both the *variable* and the *atomic proposition* associated to the variable. For example, in (7.5) r_1 and r_2 are atomic propositions that represent the truth value of corresponding variables.

Rules Additional formulas can be inferred from each specification by checking its Typeset. CROME automatically infers the following formulas:

— *Mutually Exclusion Rules* φ^{mtx} for any mutually exclusive relationship in the Typeset.

— *Refinement Rules* φ^{ref} for any refinement relationship in the Typeset.

— *Adjacency Rules* φ^{adj} for any adjacency relationship in the Typeset.

For example, let us consider the *world* W described in the previous paragraph and shown in Figure 7.1. Let $W = \{r_t, r_b, r_1, r_2, r_3, r_4, r_5, day, night\}$ be its set of types; then, the rules that can be inferred are:

$$\varphi_1^{\text{mtx}} = \bigwedge_{i=\{t,b\}} \mathbf{G}(r_i \leftrightarrow \bigwedge_{i \neq j} \neg r_j) \tag{7.6}$$

$$\varphi_2^{\text{mtx}} = \bigwedge_{i=1..5} \mathbf{G}(r_i \leftrightarrow \bigwedge_{i \neq j} \neg r_j) \tag{7.7}$$

$$\varphi_3^{\text{mtx}} = \mathbf{G}(day \leftrightarrow \neg night) \tag{7.8}$$

$$\varphi^{\text{ref}} = \mathbf{G}((r_1 \vee r_2) \rightarrow r_t) \wedge \mathbf{G}((r_3 \vee r_4) \rightarrow r_b) \tag{7.9}$$

$$\varphi^{\text{adj}} = \bigwedge_{i=1..5} \mathbf{G}(r_i \rightarrow \bigvee_{r' \in Adj(r_i)} \mathbf{X}\ (r')) \tag{7.10}$$

Where (7.6), (7.7) and (7.8) model the mutually exclusion conditions. The refinement relationships are modeled in (7.9). Finally, topology is modeled with (7.10) where $Adj(r)$ denotes the locations adjacent to r (including r itself).

To represent the requirements of the robot patrolling the locations in order we could use the robotic specification pattern *StrictOrderedPatrolling*, while to perform the actions we could use the *reaction patterns* [48]. Let **OP**, **BR** and **BD** be the *StrictOrderedPatrolling*, *BoundRection* and *BoundDelay* robotic patterns, respectively. We can define the set of specifications $\Gamma = \{\gamma_d, \gamma_n, \gamma_g\, \gamma_r\}$ where:

$$\gamma_d = \mathbf{OP}(r_1, r_2) \tag{7.11}$$
$$\gamma_n = \mathbf{OP}(r_3, r_4) \tag{7.12}$$
$$\gamma_g = \mathbf{BD}(person, greet) \tag{7.13}$$
$$\gamma_r = \mathbf{BR}(person, register) \tag{7.14}$$

Checks Any specification in CROME must be satisfiable. To check for satisfiability, CROME takes into consideration the *mutually exclusion rules* of the specification. Validity checks take into consideration any possible *refinement rules*. Finally, realizability checks take into consideration all kinds of rules: mutually exclusion, refinement, and adjacency.

Operations A new specification can be created from simpler ones via *conjunction* ($\wedge$), *disjunction* ($\vee$) and *negation* ($\neg$) of *Formula* and *Atom* elements. In each formula, we keep both the conjunctive (*'cnf'*) and disjunctive (*'dnf'*) normal forms (treating any *Atom* elements as a *literals*). For example, a disjunction between the *Atom* in (7.4) and the *Formula* in (7.5) would result in the following *Formula* where both the *'dnf'* and *'cnf'* representations are formed by two other *Formula* elements (separated by a comma) as follows:

$$\text{cnf:}\quad \mathbf{F}(r_1) \vee \mathbf{G}(s \rightarrow a), \qquad \mathbf{F}(l_2) \vee \mathbf{G}(s \rightarrow a) \tag{7.15}$$
$$\text{dnf:}\quad \mathbf{F}(r_1) \wedge \mathbf{F}(r_2), \qquad \mathbf{G}(s \rightarrow a) \tag{7.16}$$

A *Formula* can always be represented as the conjunction or disjunction of all the elements in the *cnf* or *dnf* representations, respectively.

Comparisons Specifications can be compared among each other with the operators $\leq$, $<$, $==$, $\neq$, $>$, $\geq$, where $<$ and $>$ represent the *refinement* and **abstraction** relationships, respectively. Let φ_1 and φ_2 be two specifications, we have that:

$$\varphi_1 == \varphi_2 \iff (\varphi_1 \rightarrow \varphi_2 \wedge \varphi_2 \rightarrow \varphi_1) \text{ is } \textit{valid} \tag{7.17}$$
$$\varphi_1 \neq \varphi_2 \iff \neg(\varphi_1 == \varphi_2) \tag{7.18}$$
$$\varphi_1 \leq \varphi_2 \iff (\varphi_1 \rightarrow \varphi_2) \text{ is } \textit{valid} \tag{7.19}$$
$$\varphi_1 \geq \varphi_2 \iff (\varphi_2 \rightarrow \varphi_1) \text{ is } \textit{valid} \tag{7.20}$$
$$\varphi_1 < \varphi_2 \iff (\varphi_1 \leq \varphi_2) \wedge (\varphi_1 \neq \varphi_2) \tag{7.21}$$
$$\varphi_1 > \varphi_2 \iff (\varphi_1 \geq \varphi_2) \wedge (\varphi_1 \neq \varphi_2) \tag{7.22}$$

Every time a check, a comparison, or an operation among specifications is performed, new constraints are extracted from the *Typeset* of each specification involved. Specifically, CROME creates a new *Typeset* containing the *union* of all the *types* in all the typeset and extracts a new set of refinement, mutually exclusion, and adjacency relationships.

Contract A contract $\mathcal{C} = (\psi, \phi)$ is a tuple of specifications, where ψ represents the assumptions and ϕ represents the guarantees. The guarantees in a contract are always considered in their saturated form, i.e. having guarantees $\psi \rightarrow \phi$. If a contract gets its assumptions changed or updated, its guarantees will change as well due to saturation.

For our running example, we can formalize three contracts:

$$\mathcal{C}_d = (true, \mathbf{OP}(r_1, r_2)) \tag{7.23}$$
$$\mathcal{C}_n = (true, \mathbf{OP}(r_1, r_2)) \tag{7.24}$$
$$\mathcal{C}_r = (\varphi_s^{\text{fair}}, \mathbf{BR}(person, register)) \tag{7.25}$$
$$\mathcal{C}_g = (\varphi_s^{\text{fair}}, \mathbf{BD}(person, greet)) \tag{7.26}$$

Where φ_s^{fair} represents a *fairness* assumption on the sensor variables, i.e. $\mathbf{G}(\mathbf{F}(person))$, since we expect to see a person infinitely often.

Checks Contracts in CROME are always *compatible*, *consistent*, and *feasible*. Given a contract $\mathcal{C} = (\psi, \phi)$, we have that:

$$\mathcal{C} \text{ is compatible} \iff \psi \text{ is } \textit{satisfiable} \tag{7.27}$$
$$\mathcal{C} \text{ is consistent} \iff \phi \text{ is } \textit{satisfiable} \tag{7.28}$$
$$\mathcal{C} \text{ is feasible} \iff (\psi \wedge \phi) \text{ is } \textit{satisfiable} \tag{7.29}$$

Operations CROME supports the operations of *composition* and *conjunction* among contracts. Let $\mathcal{C}_1 = (\psi_1, \phi_1)$ and $\mathcal{C}_2 = (\psi_2, \phi_2)$ be two contracts.

The *composition* $\mathcal{C}_1 \mid\mid \mathcal{C}_2$ results in a contract $\mathcal{C}^{||} = (\psi^{||}, \phi^{||})$ such that:

$$\psi^{||} = \psi_1 \wedge \psi_2 \vee \neg(\phi_1 \wedge \phi_2) \tag{7.30}$$
$$\phi^{||} = \phi_1 \wedge \phi_2 \tag{7.31}$$

The *conjunction* $\mathcal{C}_1 \wedge \mathcal{C}_2$ results in a contract $\mathcal{C}^{\wedge} = (\psi^{\wedge}, \phi^{\wedge})$ such that:

$$\psi^{\wedge} = \psi_1 \vee \psi_2 \tag{7.32}$$
$$\phi^{\wedge} = \phi_1 \wedge \phi_2 \tag{7.33}$$

Comparisons As with the specification, we can compare contracts. Let us consider the contracts $\mathcal{C}_1 = (\psi_1, \phi_1)$ and $\mathcal{C}_2 = (\psi_2, \phi_2)$; we have that:

$$\mathcal{C}_1 == \mathcal{C}_2 \iff (\psi_1 == \psi_2) \wedge (\phi_1 == \phi_2) \tag{7.34}$$
$$\mathcal{C}_1 \neq \mathcal{C}_2 \iff (\psi_1 \neq \psi_2) \wedge (\phi_1 \neq \phi_2) \tag{7.35}$$
$$\mathcal{C}_1 \leq \mathcal{C}_2 \iff (\psi_1 \geq \psi_2) \wedge (\phi_1 \leq \phi_2) \tag{7.36}$$
$$\mathcal{C}_1 \geq \mathcal{C}_2 \iff (\psi_1 \leq \psi_2) \wedge (\phi_1 \geq \phi_2) \tag{7.37}$$
$$\mathcal{C}_1 < \mathcal{C}_2 \iff (\mathcal{C}_1 \leq \mathcal{C}_2) \wedge (\mathcal{C}_1 \neq \mathcal{C}_2) \tag{7.38}$$
$$\mathcal{C}_1 > \mathcal{C}_2 \iff (\mathcal{C}_1 \geq \mathcal{C}_2) \wedge (\mathcal{C}_1 \neq \mathcal{C}_2) \tag{7.39}$$

Goal In CROME, each mission requirement is modeled by a goal, which is characterized by the following elements:

- *Name*: goal identifier;
- *Description*: English description of the mission requirement;
- *Context*: Specification containing a Boolean predicate on type variables of kind 'context';
- *Contract*: a Contract object formalizing the objective of the goal.
- *Controller*: Mealy machine realizing the contract associated to the goal.
- *World*: *World* object where the goal will be deployed.

Contexts are formalized in terms of Boolean predicates encoding the situation in which a goal must be active. For example, context propositions can encode information related to the location, time, or identities associated with a goal. In a robotic application, locations specify *where* a robot can be, time specifies *when* a certain goal must be active (e.g., during the day or the night), and identities specify the state of external entities (*who*) that may interact with the robot.

Contracts can be build up from specification using, for example, robotic patterns to indicate what is the objective of the robot in the goal being modeled. There is always only one contract associated with each goal. We will refer the contract belonging to the goal $\mathcal{G}_i$ with the contract $\mathcal{C}_i$.

Controllers are generated by realizing the specification associated with the Contract. Specifically, if the guarantees of the contract $\psi \rightarrow \phi$ is a specification realizable in a controller, then a finite state machine that satisfies the specification is produced, i.e. the *controller*.

The *world* object is needed to extract all the refinement, adjacency, and mutually exclusion rules that the robot is subject to in the mission. Such rules will add constraints on the robot behavior and, according to the world configuration, could jeopardize the satisfiability or realizability of the missions.

Checks, Comparisons, and Operations We reduce all the formal operations, checks, and comparisons among goals to their corresponding contracts. Goal conflicts are detected by checking the compatibility, consistency, and feasibility of the contract formalizing the goals. CROME links the conflicting clauses in the specifications of the contract to the goals that generated them and presents these goals to the designer.

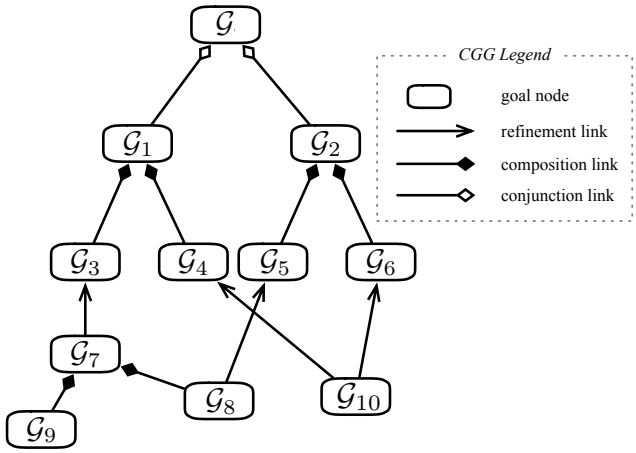


Figure 7.3: Example of a CGG.

CGG Each goal is linked to one node of the CGG (Contract-based Goal Graph), which is a formal model that represents the goals in a hierarchical way. A CGG, shown in Figure 7.3, is a graph $T = (\Upsilon, \Xi)$, where each node $\upsilon \in \Upsilon$ is a goal and each *directional* edge $\xi \in \Xi$ is a link that connects two goals together. A *Link* among goals can be of three kinds: *refinement*, *composition* and *conjunction*. A refinement link connects a goal to a more abstract one. The semantic associated with the other kinds of links is the following: all the incoming edges of the same kind of link in a node n represent the goal nodes contributing to generating n and the operation involved according to the kind of link, i.e. composition or conjunction.

For example, Figure 7.3 shows an example of the CGG where the top level node, i.e. goal $\mathcal{G}$, is the result of the *conjunction* between $\mathcal{G}_1$ and $\mathcal{G}_2$. Where $\mathcal{G}_1$ is the *composition* of goals $\mathcal{G}_3$ and $\mathcal{G}_2$ while $\mathcal{G}_1$ is the *composition* of goals $\mathcal{G}_5$ and $\mathcal{G}_6$. A goal node can have multiple outgoing edges, like goal $\mathcal{G}_{10}$, which *refines* both $\mathcal{G}_4$ and $\mathcal{G}_6$. Finally, we have that $\mathcal{G}_7$, which refined $\mathcal{G}_3$, is composed of $\mathcal{G}_9$ and $\mathcal{G}_8$. Notice that $\mathcal{G}_8$ besides participating in the *composition* of $\mathcal{G}_7$ also refines $\mathcal{G}_5$.

Refinements links can be produced manually or automatically. Automatic refinements can be produced by searching in a library of goals if any composition *refines* the specification of any leaf-node of the CGG, as in the case of goal $\mathcal{G}_9$ and $\mathcal{G}_8$ in Figure 7.3. We show how refinements from the library can be performed in [200, 222]. Whenever a refinement link is created, the assumptions resulting from the more refined goal will be automatically propagated in the more abstract goal and, by following the edges of the CGG, up to the root as detailed in [222]. Assumption propagation makes sure that the refinements are always consistent, i.e., the set of assumptions of the refined goal must always be greater than or equal to those of the abstract goal.

Figure 7.4 shows the CGG that CROME automatically created according to our *context-based specification clustering* algorithm presented in [200] given three goals as input $\mathcal{G}_n$, $\mathcal{G}_d$ and $\mathcal{G}_g$ which are the goal related to the contracts formalized in (7.24), (7.23) and (7.26). The composition of $\mathcal{G}_d$ with $\mathcal{G}_g$ and $\mathcal{G}_r$

(i.e. $\mathcal{G}_{dgr}$) and $\mathcal{G}_n$ with $\mathcal{G}_g$ (i.e. $\mathcal{G}_{ng}$) produce the following contracts:

$$\mathcal{C}_{dgr} = (\psi_{dgr}, \phi_{dgr}) \tag{7.40}$$
$$\mathcal{C}_{ng} = (\psi_{ng}, \phi_{ng}) \tag{7.41}$$
$$\tag{7.42}$$

Where:

$$\psi_{dgr} = \varphi_s^{\text{fair}} \vee \neg\phi_{dgr} \tag{7.43}$$
$$\phi_{dgr} = \mathbf{OP}(r_1, r_2) \wedge (\varphi_s^{\text{fair}} \rightarrow \mathbf{BR}(p, r)) \wedge (\varphi_s^{\text{fair}} \rightarrow \mathbf{BD}(p, g)) \tag{7.44}$$
$$\psi_{ng} = \varphi_s^{\text{fair}} \vee \neg\phi_{ng} \tag{7.45}$$
$$\phi_{ng} = \mathbf{OP}(r_3, r_4) \wedge (\varphi_s^{\text{fair}} \rightarrow \mathbf{IR}(p, g)) \tag{7.46}$$

The goals $\mathcal{G}_{ng}$ and $\mathcal{G}_{dgr}$ represent the two *scenarios* of the mission. Specifically, they represent what the robot should do when the context is *night* or *day*, respectively. The controllers λ_1 and λ_2 are generated via reactive synthesis from the contracts $\mathcal{C}_{ng}$ and $\mathcal{C}_{dgr}$, respectively. Specifically, we define the set of specification $\Gamma = [\gamma_1, \gamma_2]$ where:

$$\gamma_1 = \psi_{dgr} \rightarrow \phi_{dgr} \tag{7.47}$$
$$\gamma_2 = \psi_{ng} \rightarrow \phi_{ng} \tag{7.48}$$

We have that λ_1 and λ_2 are finite state machines that satisfy γ_1 and γ_2, respectively. By synthesizing automatically a controller also for the root node of the CGG, we would obtain a controller that satisfies the specification but does not satisfy the mission. In fact, there is no notion of *context* inside the specification and our *contextual* mission cannot be satisfied. In the following, we formally define what does it mean to satisfy a contextual mission. Then, we show the problems of defining the contextual mission only using our previous version of CROME, which used only LTL. Finally, we show how we tackle these problems and ultimately generate a controller that both satisfies the CGG specification and the contextual mission.

7.4 From Mission Specification to Mission Controller

Controller synthesis is the problem of generating a correct controller for the robotic-system that acts according to what it is prescribed by the specification, guaranteeing that the controller has a strategy to achieve its goal whatever the environment does. A controller dictates a *policy* to the system, i.e. for every environment configuration, it restricts what action the system should perform such that it would not violate its requirements.

In [200] we generate a system controllers using *reactive synthesis*. As the mission specification becomes larger, the controller can become infeasible to

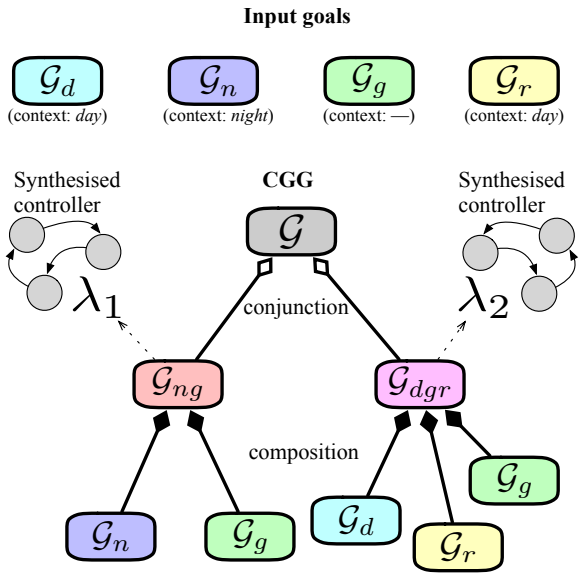


Figure 7.4: CGG created for our running example.

realize. That is one of the reasons why, in [200], we have separated the overall-mission in mutually-exclusive *scenarios*, which can be more manageable to realize with reactive synthesis. Scenarios are partial missions that should be achieved under a certain context. The effect of this is to reduce each mission to one and only one of such mutually exclusive scenarios, i.e. there is no possibility to switch from one scenario to another as the context changes. In other words, we can produce a controller for each of the scenarios assuming that the mission will happen in either one of them, i.e. there is not context-change throughout the whole mission. However, this assumption is too restrictive. In fact, often, during the mission execution the contexts can change, and so the robotic controller must adapt to the new scenario [175].

A *context change* is an uncontrollable event and it can happen at any point in time during the execution of the mission. At all times we have a *scenario* that is active and so a controller that is executing some action. When a change of context happens, e.g. context x_1 to context x_2, we would like the robot controller to switch from Λ_1 to Λ_2, where Λ_1 and Λ_2 are the controllers synthesized from the scenarios of context x_1 and x_2, respectively. We would also like that when x_1 becomes the current context again, the control switches back to Λ_1 by completing or resuming the task from where it was halted before.

In [200] we have produced the specification for the mission scenarios only using LTL. Our formalization was an oversimplification of the problem since the LTL formula did not capture the possibility for the context to change during the mission. In the following:

— we formally define what is the problem of realizing a contextual mission. Having a formal definition allows us to prove if our controllers achieve the contextual mission under our assumptions.

— we present why it is hard to express contextual missions only using LTL

and argue for a different approach.

— we present our solution to formally realize contextual missions.

7.4.1 Problem Definition

Let n be the number of contexts in a mission.

A **world** is a tuple $\mathcal{E} = (\Omega_S, \Omega_R, \Omega_A, \gamma_e)$, where:

— Ω_S is a set of Boolean variables representing the robot *sensors*.

— Ω_R is a set of Boolean variables representing the *regions* in a certain environment.

— Ω_A is a set of Boolean variables representing the *actions* that a robot can perform.

— γ_e is an LTL formula expressed on $\mathcal{AP}_s \cup \mathcal{AP}_r \cup \mathcal{AP}_a$, where we have defined the following atomic propositions:

  – s for each sensor $s \in \Omega_S$(e.g. human is true if and only if a human is detected by the sensor).

  – r for each region $r \in \Omega_R$ (e.g. r_1 is true if and only if the robot is in location r_1).

  – a for each action $a \in \Omega_A$ (e.g. greet is true if and only if a the robot is greeting).

  $\mathcal{AP}_s, \mathcal{AP}_r$ and $\mathcal{AP}_a$ are the sets of all atomic propositions s, r, and a, respectively.

Intuitively, γ_e represents a set of constraints imposed by the world where the robot is operating.

A **mission** (or *contextual mission*) is a tuple $\mathcal{M} = (X, \Gamma, AC, CT)$. Where:

— $X = \{x_1, x_2, ..., x_n\}$ is the set of *contexts* where any $x_i \in X$ is a boolean proposition.

— $\Gamma = \{\gamma_1, \gamma_2, ..., \gamma_n\}$ is the set of *specifications* where any $\gamma_i \in \Gamma$ is an LTL formula defined over a set of atomic propositions $\mathcal{AP}_i = \mathcal{I}_i \cup \mathcal{O}_i$ and where $\mathcal{I}_i \subseteq \mathcal{AP}_s$ and $\mathcal{O}_i \subseteq \{\mathcal{AP}_r \cup \mathcal{AP}_a\}$.

— $AC \colon \mathbb{N} \to X$ is a function that given a time $t \in \mathbb{N}$ as input returns which context $x_i \in X$ is *active* at that time.

— $CT : X \to \Gamma$ is a function that, for any context $x_i \in X$, returns a specification $\gamma_i \in \Gamma$.

A **robot** is a finite-state machine $\mathcal{R} = (S, \mathcal{I}, \mathcal{O}, s_0, \delta)$ where:

— S is the set of states.

— $s_0 \in S$ is the initial state.

— $\mathcal{I} = \mathcal{AP}_s \cup \mathcal{I}_X$ is the set of input propositions, where $\mathcal{I}_X = \{x_1, x_2, ..., x_n\}$ is a set of atomic propositions, where each x_i represents the context $x_i \in X$.

— $\mathcal{O} = \mathcal{AP}_r \cup \mathcal{AP}_a \cup \mathit{Active}$ is the set of output propositions, where *Active* is a Boolean signal that determines whether the robot is *active*.

— $\delta : S \times 2^{\mathcal{I}} \to S \times 2^{\mathcal{O}}$ is the transition function.

We define:

— A **context switch**: If $x_i \in X$ is active at time $t-1$ and $x_j \in X$ is active at time t, with $i \neq j$, we say that a *context switch* happened at time t between context x_i and context x_j.

— A **robot trace** $\mathit{Trace}^{\mathcal{R}}$*:* we say that some word $w = (w_0^{\mathcal{I}}, w_0^{\mathcal{O}})(w_1^{\mathcal{I}}, w_1^{\mathcal{O}})(w_2^{\mathcal{I}}, w_2^{\mathcal{O}})... \in (2^{\mathcal{I}} \times 2^{\mathcal{O}})^{\omega}$ is a *trace* of $\mathcal{R}$ if there exists a run $\tau = \tau_0\tau_1\tau_2... \in S^{\omega}$ such that $\tau_0 = s_0$ and for every $i \in \mathbb{N}$ we have that $(\tau_{i+1}, w_i^{\mathcal{O}}) = \delta(\tau_i, w_i^{\mathcal{I}})$. We indicate with $\omega_t = (\omega_t^{\mathcal{I}}, \omega_t^{\mathcal{O}})$ the trace (i.e. the input and output value) at the instant of time t.

— A **trace projection** $TP_{\gamma_i}^{\mathcal{R}}$ given a robot $\mathcal{R}$ on a specification γ_i, is the words in $\mathit{Trace}^{\mathcal{R}}$ at the time-steps $t \in \mathbb{N}$ where the specification related to the active context in t is γ_i and *Active* is true. That is, given a word $w = a_0, a_1, \ldots \in \mathit{Trace}^{\mathcal{R}}$, we have $pr_{\gamma_i}(w) = a_{i_1}, a_{i_1}, \ldots$ where $i_0, i_1, \ldots$ are all the times t such that $CT(AC(t)) = \gamma_i$. Then, $TP_{\gamma_i}^{\mathcal{R}} = \{pr_{\gamma_i}(w) \mid w \in \mathit{Trace}^{\mathcal{R}}\}$.

We assume that:

— Any specification $\gamma_i \in \Gamma$ is realizable.

— Contexts are mutually exclusive, i.e. any $x_i \wedge x_j$ is not satisfiable, with $x_i \in X$, $x_j \in X$ and $i \neq j$.

— Any context $x_i \in X$ is active infinitely often, i.e. $\mathbf{G}(\mathbf{F}(x_i))\forall x_i \in X$.

— Any context is active for at least $t_{\texttt{context}}$ consecutive time units.

Definition 7.4.1 (Contextual Mission Satisfaction)**.** A robot $\mathcal{R}$ satisfies a mission $\mathcal{M}$ in the world $\mathcal{E}$, i.e. $\mathcal{R}, \mathcal{E} \models \mathcal{M}$, if and only if:

— Each trace projection satisfies the corresponding specification, i.e. $\forall \gamma_i \in \Gamma$, $TP_{\gamma_i}^{\mathcal{R}} \models \gamma_i$.

— *Active* becomes *false* at any context-switch and stays *false* for at most $t_{\texttt{trans}}$ time units, where $t_{\texttt{trans}} \leq t_{\texttt{context}}$.

Figure 7.5 shows an example of active contexts and tasks between $t = 0$ and $t = 10$, with the overall trace of the robot $\mathit{Trace}(\mathcal{R})$ and its trace projections over the specifications γ_1 and γ_3. For some world $\mathcal{E}$, we have that $\mathcal{R}, \mathcal{E} \models \mathcal{M}$ if:

— $TP_{\gamma_1}^{\mathcal{R}} = w_5, w_6 \ldots \models \gamma_1$ and

— $TP_{\gamma_3}^{\mathcal{R}} = w_0, w_1, w_2, w_8, w_9, w_{10} \ldots \models \gamma_3$.

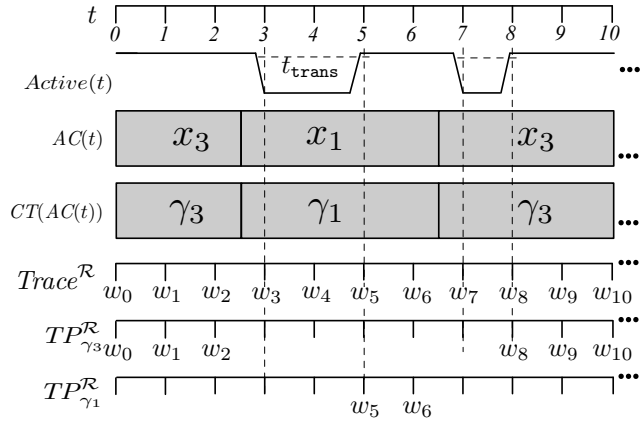


Figure 7.5: Example of two contextual tasks active at different points in time.

7.4.2 Difficulties by using only LTL Contracts to define the mission

We could formulate a mission as a single contract $\mathcal{C} = (\psi, \phi)$, where ψ and ϕ are LTL formulae and automatically synthesize a robot that satisfies $\psi \rightarrow \phi$. However, formulating the mission as a pair of LTL formulae is not easy. Specifically, given $\mathcal{E}, \mathcal{M}$, can we produce a contract $\mathcal{C}$ that describes the set of traces to be satisfied by the robot as we have defined them in (7.4.1)?

For example, we could think of defining a mission as the conjunction of *contextual tasks*, where each contextual task must be satisfied under a specific context i.e. $M = \{(x_1, \gamma_1), (x_2, \gamma_2), ..., (x_n, \gamma_n)\}$ where each $M_i = (x_i, \gamma_i) \in M$ is a contextual task, $x_i \in X$ and $\gamma_i \in \Gamma$. In the following, we see some of the problems that we can encounter while trying to formalizing contextual tasks and the overall mission specification as described in Section 7.4.1 using our previous version of CROME, which used only LTL contracts.

Problem 1: 'Expressing contextual tasks with a simple implication' For some γ_i we can think of expressing contextual tasks using a simple *instant reaction* robotic pattern [48] that 'applies when the occurrence of a stimulus instantaneously triggers a counteraction'. Given a context x_1 and a specification γ_1 we can express the behavior of a contextual task with the following contract:

$$\mathcal{C} = (\mathit{true}, \mathbf{G}(x_1 \rightarrow \gamma_1)) \tag{7.49}$$

For $\gamma_1 = a$, where a is an atomic proposition, we have that for every step in which the context x_1 is true, γ_1 must be true, hence the semantics of the contextual task is preserved. Also, in case of $\gamma_1 = \mathbf{X}a$ or $\gamma_1 = \mathbf{F}a$, we have that the context x_1 enforces some behaviors to happen in γ_1. That is, a must be *true* the next step every time x_1 is *true* and a must be *true* eventually whenever x_1 has been observed to be *true*, respectively.

However, if γ_1 is an LTL formula that must hold *always* (e.g. $\gamma_1 = \mathbf{G}(a)$ or $\gamma_1 = \mathbf{GF}(a)$), then the semantics of contextual task, as it is informally described above, becomes meaningless, i.e. γ_1 is satisfied regardless of the value

of the context x_1, or x_1 is never true, which violates one of our assumption of a context being *true* infinitely often.

Problem 2: 'Modeling the task switch and context duration intervals' Let us consider two simple contextual tasks expressed by the contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ over contexts x_1 and x_2 and locations r_1 and r_3:

$$\mathcal{C}_1 = (true, \mathbf{G}(x_1 \rightarrow \mathbf{X}r_1)) \tag{7.50}$$
$$\mathcal{C}_2 = (true, \mathbf{G}(x_2 \rightarrow \mathbf{X}r_3)) \tag{7.51}$$

Their joint behavior can be modeled with the contract $\mathcal{C}$:

$$\mathcal{C} = \left(\varphi_{\mathrm{c}}^{\mathrm{mtx}}, \quad \phi_1 \wedge \phi_2 \wedge \varphi_{\mathrm{r}}^{\mathrm{mtx}}\right) \tag{7.52}$$

Assuming that all locations are mutually exclusive with each other (i.e. $\varphi_{\mathrm{r}}^{\mathrm{mtx}}$), and the contexts x_1 and x_2 are also mutually exclusive (i.e. $\varphi_{\mathrm{c}}^{\mathrm{mtx}}$), then a controller for the specification exists. However, without any assumption on the minimum duration of a context, we might easily have unrealizable specifications such as:

$$\mathcal{C}_1' = (true, \mathbf{G}(x_1 \rightarrow (\mathbf{X}r_1 \wedge \mathbf{XX}r_2)) \tag{7.53}$$
$$\mathcal{C}_2' = (true, \mathbf{G}(x_2 \rightarrow (\mathbf{X}r_3 \wedge \mathbf{XX}r_4)) \tag{7.54}$$

In fact, a context might change at each step forcing the specification to be violated.

Modeling a minimum *context-switch* interval in LTL, i.e. the minimum number of time steps where one context has to hold, is not straightforward. For example, by simply adding the assumption $\mathbf{G}(x_1 \rightarrow \mathbf{X}x_1)$ will make the context x_1 true forever after the first time it becomes true, thus violating the assumption that contexts become true infinitely often and preventing the specification to become realizable.

Problem 3: 'Topologies might add delays' According to the topology of the world in which the robot is operating, it is might be impossible to switch from one task to another in one step. Let us consider the two contextual tasks formalized as in (7.53) and (7.54). Let us instantiate these tasks in the world depicted in Figure 7.1. We have that in order to go from, for example, location r_2 to r_3 the robot would need to *pass by* another location (i.e. r_5). The specification in the contracts $\mathcal{C}_1'$ or $\mathcal{C}_2'$ prescribes the robot to reach the locations r_1 or r_3 one step after the context has become true. However, if the robot has to pass by r_5, it would require an extra time step making the specification unrealizable.

Problem 4: 'Context Switch and Memory' Whenever there is a context switch, the robot should start satisfying the specification active under the context from the last visited state. Let us look at Figure 7.5, we have that the

sequence $w_0, w_1, w_2, w_3, w_8, w_9, w_{10}$ must satisfy the specification γ_3. We can note that there is a *jump* between w_3 and w_8 since the two words happen at timestep 3 and 8, respectively. However, the time gap between the timesteps 3 and 8 is not relevant for the satisfaction of γ_3. That means that whenever the robot is satisfying a task γ_i, it is able to halt its execution at timestep 4 and resume later at timestep 8.

Let us consider the example described in Section 7.1.2 and depicted in Figure 7.1. Both contextual tasks (i.e. the specifications be performed during the context day and night respectively) indicate to the robot to start patrolling from location r_1 or r_3. During the day it goes back and forward between r_1 and r_2 and during the night between r_3 and r_4. If in the transition from the context day and the context night the robot was visiting r_1, when the context day becomes active again, the next visit location should be r_2 and not r_1. In order to encode the memory of the current and the last visited location in each task, we would need to add extra output variables and modifying all the specifications to write and read from such variables. This is not practical to do.

7.5 Controllers Orchestration

We present our approach to tackle the problem defined in Section 7.4.1 considering the example in Section 7.1.2. In [200] we have incorporated the context inside the LTL mission specification and automatically synthesized a controller out of the full specification. However, with this approach, we are not able to generate a controller that satisfies the contextual mission (7.4.1). In this paper, we do not include the contexts inside the LTL mission specification. Instead, by using external machinery we are able to orchestrate the different specification controllers in a way that also the overall contextual mission is satisfied.

By synthesizing automatically a controller also for the root node of the CGG, we would obtain a controller that satisfies the specification but does not satisfy the mission. In fact, there is no notion of *context* inside the specification and our *contextual* mission cannot be satisfied according to our definition. In the following, we show how we tackle this problem and ultimately generate a controller that both satisfies the CGG specification and the contextual mission.

7.5.1 Specification and Transition Controllers

Definition 7.5.1 (Specification Controllers)**.** Let $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ be the set of *specification controllers*, where for all $1 \leq i \leq n$, controller $\lambda_i \in \Lambda$ realizes the corresponding specification $\gamma_i \in \Gamma$.

Every specification $\gamma_i \in \Gamma$ can either indicate the robot to move among regions or activate some action. While actions can be activated at any point in time regardless of the robot's current location, moving from one region to another can add additional constraints according to the topology of the environment as we have seen in *problem 3* of Section 7.4.1. Besides controllers realizing the specifications, we need additional controllers that are responsible for guiding the robot so that it can reach the locations where it can satisfy the specification.

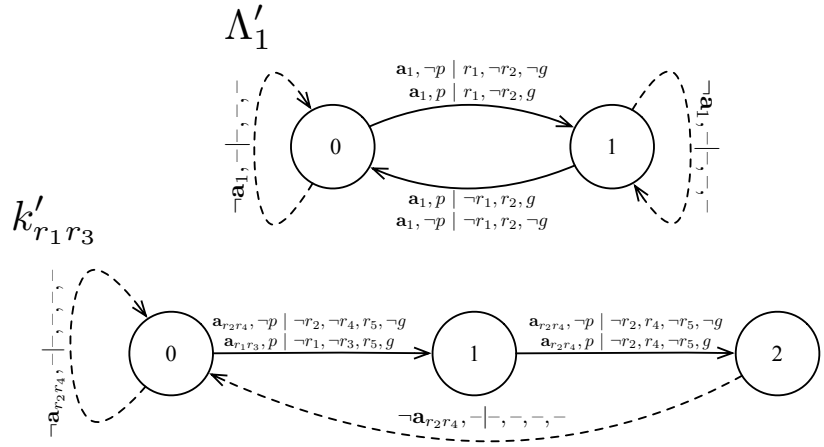


Figure 7.6: A specification and transition controllers that have been modified with the activation inputs.

For example, considering the illustrative example, when the contexts switch from day to night, the robot must switch patrolling locations r_1 and r_2 to patrolling locations r_3 and r_4. However, immediately after the context switch happens, there is a *transition phase* in which the robot needs to be guided by a controller that brings it from any locations in $\{r_1, r_2\}$ to any locations in $\{r_3, r_4\}$. Since a context switch can happen while the robot is operating in any location, we need a controller for any pair of locations: that is from any location to any other location.

Definition 7.5.2 (Transition controllers)**.** We define a *transition controller* k_{r_i,r_j} to be a controller that starting from r_i reaches r_j in the least number of steps, where r_i and $r_j \in \Omega_R$ and $i \neq j$. We define K to be the set of all possible transition controllers, i.e. a controller for each pair of locations $r_i, r_j \in \Omega_R$. That is $K = \{k_{r_i r_j} \forall r_i, r_j \in \Omega_R : i \neq j\}$.

Definition 7.5.3 (Transition time)**.** We define the transition time, $t_{\texttt{trans}}$, to be the maximum amount of time units needed for any controller $k_{r_i,r_j} \in K$ to start from location r_i and reaching location r_j.

Active controllers CROME modifies the specification and transition controller in order to define when a controller is *active* and when it is *inactive*. Controllers are finite state machines (more specifically, *Mealy machines*) defined over a set of input and output propositions. We define two new sets of atomic propositions:

$$\mathcal{I}^{\mathbf{a}}_{\Lambda} = \{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n\}$$
$$\mathcal{I}^{\mathbf{a}}_{K} = \{\mathbf{a}_{r_i r_j} \forall r_i, r_j \in \Omega_R : i \neq j\}$$

CROME modifies the controllers in Λ and K by adding an atomic proposition in $\mathcal{I}^{\mathbf{a}}_{\Lambda}$ and K as additional controllers inputs and we refer to them as *activation*

inputs. Specifically, we add an atomic proposition $\mathbf{a}_i \in \mathcal{I}_\Lambda^{\mathbf{a}}$ for any specification controller $\lambda_i \in \Lambda$ and $\mathbf{a}_{r_i r_j}$ for any transition controller $k_{r_i r_j} \in K$. By controlling the activation inputs CROME is able to *orchestrate* the reactions happening in the controllers as we show later in Section 7.6.

For any specification controller $\lambda_i \in \Lambda$ we add a *self-loop* transition for any state of λ_i. This added transition accepts as input the negation of its activation input (i.e. $\neg\mathbf{a}_i$) and leaves the outputs to be undetermined (–).

For any transition controller $\lambda_i \in \Lambda$ we add a *self-loop* transition to its initial state and another transition that goes from its terminal state to its start state. The rationale behind this choice is that while a specification controller can be deactivated and activated in any state, a transition controller once activated terminates its execution reaching its final state. As before, these transitions accept as input the negation of its activation input (i.e. $\neg\mathbf{a}_{r_i r_j}$) for any input of $k_{r_i r_j}$ and leaving the output to be undetermined (–). We define $\Lambda' = \{\lambda'_1, \lambda'_2, ..., \lambda'_n\}$ and $K' = \{k'_{r_i r_j} \forall r_i, r_j \in \Omega_R : i \neq j\}$ to be the specification and transition controller after the modification described above.

If any controller $\lambda'_i \in \Lambda'$ or $k'_{r_i r_j} \in K'$ has its activation input set to *true* we say that λ'_1 or $k'_{r_i r_j}$ is *active*, otherwise, we say that the controller is *inactive*. Figure 7.6 shows two controllers that have been modified by CROME. The specification controller Λ_i has inputs $\mathbf{a}_1, p$, outputs r_1, r_2, g, start state 0 and no terminal state. The transition controller $k_{r_1 r_3}$ has inputs $\mathbf{a}_{r_1 r_3}, p$, outputs r_1, r_3, r_5, g, start state 0 and terminal state 3. CROME has added the *activation inputs* $\mathbf{a}_1$ and $\mathbf{a}_{r_1 r_3}$ and the transitions that are represented with dashed lines in Figure 7.6. These two controllers are generated from the specification resulting in the composition of the goals $\mathcal{G}_d$ and $\mathcal{G}_g$, i.e. patrol during the day and greet immediately when a person is detected.

Remark. The traces produced by any $\lambda'_i \in Lambda'$ are equivalent to the traces produced by $\lambda_i \in Lambda$ when λ'_i is *active*. Similarly, for the traces produced by any $k'_{r_i r_j} \in K'$ are equivalent to the traces produced by $k_{r_i r_j} \in K$ when $k'_{r_i r_j}$ is *active*.

Composition Given a set of controllers $\Lambda^* = \Lambda' \cup K'$ where each element $\lambda^i \in \Lambda^*$ is a finite state machine $\lambda^i = (S^i, \mathcal{I}^i, \mathcal{O}^i, s_0^i, \delta^i)$, $N = |\Lambda^*|$ and where in_t^i and out_t^i are respectively the inputs and outputs of λ^i at time t. We indicate with subscripts s_0^i, s_t^i and s_{t+1}^i the state of λ^i at time 0, t and $t+1$, respectively. We define the composition of all the elements in Λ^* as a finite-state machine $\Lambda^{||} = (S^{||}, \mathcal{I}^{||}, \mathcal{O}^{||}, s_0^{||}, \delta^{||})$ where:

$$
\begin{aligned}
S^{||} &= S^1 \times S^2 \times ... \times S^N \\
\mathcal{I}^{||} &= \mathcal{I}^1 \times \mathcal{I}^2 \times ... \times \mathcal{I}^N \\
\mathcal{O}^{||} &= \mathcal{O}^1 \times \mathcal{O}^2 \times ... \times \mathcal{O}^N \\
s_0^{||} &= (s_0^1, s_0^2, ..., s_0^N) \\
\delta^{||} &: S^{||} \times \mathcal{I}^{||} \to S^{||} \times \mathcal{O}^{||}
\end{aligned}
$$

Where we have that the transition function $\delta^{||}$ consists of the transition

functions of all the component machines side-by-side, that is:

$$((s^1_{t+1}, s^2_{t+1}, ..., s^N_{t+1}), (out^1_t, out^2_t, ..., out^N_t)) \\ = \delta^{||}((s^1_t, s^2_t, ..., s^N_t), (in^1_t, in^2_t, ..., in^N_t))$$

where $(s^i_{t+1}, out^i_t) = \delta^i(s^i_{t+1}, in^i_t)$ for $i = \{1..N\}$.

Each state machine in the composition reacts simultaneously and instantaneously. A reaction in their composition consists of a set of reactions of each state machine. Each machine $\lambda^i \in \Lambda*$ will set some outputs $\mathcal{O}^i \in \mathcal{O}^{||}$ according to some inputs $\mathcal{I}^i \in \mathcal{O}^{||}$. This kind of composition is referred in the literature as *side-by-side* composition [223]. Note that we do not include *stuttering* input and output, hence a machine in the composition can not proceed while another machine stutters. All machines proceed together. In order for the composition to be *well-defined* we assume that:

— There are no *conflicting-outputs*. That for any two machines $\lambda_i, \lambda_j \in \Lambda*$ where $i \neq j$ and where $\mathcal{O}^i \cap \mathcal{O}^j$, then $\forall t \in \mathbb{N}$ we have that their output assignments are not conflicting, i.e. $out^i_t \wedge out^j_t$ is satisfiable.

7.5.2 The orchestration system

The orchestration system monitors which context $x_i \in X$ is active and orchestrates the behavior of the finite state machine $\Lambda^{||}$ such that it satisfies the mission. The orchestration system controls the activation inputs of $\mathcal{O}^{||}$ and determines all its output. If at any $t \in \mathbb{N}$ any output symbol $o_t \in \mathcal{O}^{||}$ is not determined (–) by any machine $\lambda_i \in \Lambda*$ then it assigns $o_t = \mathit{false}$. We have modeled the orchestration system as a network of timed automata.

Timed automata are fined state machines extended with clock variables. However, we will not make explicit use of clocks that use real-time, but rather use a standard simulation of discrete-time, i.e. ticks. A template is an automaton model that can be instantiated in several automata. Figure 7.7 shows the templates for the different systems engaging in the orchestration. Transitions can be labeled with synchronization channels, guards, or updates. Predicates indicated in Figure 7.7 next to states are state *invariants* (i.e. conditions that must be respected while the automaton is in the state). Updates can be simple assignments or entire functions. We use $name := value$ to indicate assignment, to distinguish it from a predicate ($name = value$).

Template instances There are n instances of *Context* template, each modeling one context. We have $(n^2 - n)$ instances of the *T-Controller* template, each modeling a *transition controller* from context x_i to a context x_j where $i \neq j$. For each *specification controller* we have associated an instance of the *S-Controller* template. There is always a specification controller for any context. Finally the *Orchestrator* automaton coordinates the states of all the instances of *Context*, *S-Controller* and *T-Controller* templates. In the following we will refer to contexts, specification controllers and transition controllers to indicate the instances of the corresponding templates. In particular we will use the abbreviated notation $C[i]$, $S[i]$ and $T[i][j]$ to indicate specific instances contexts, specification and transition controllers. Finally we will use O to indicate the orchestrator automaton.

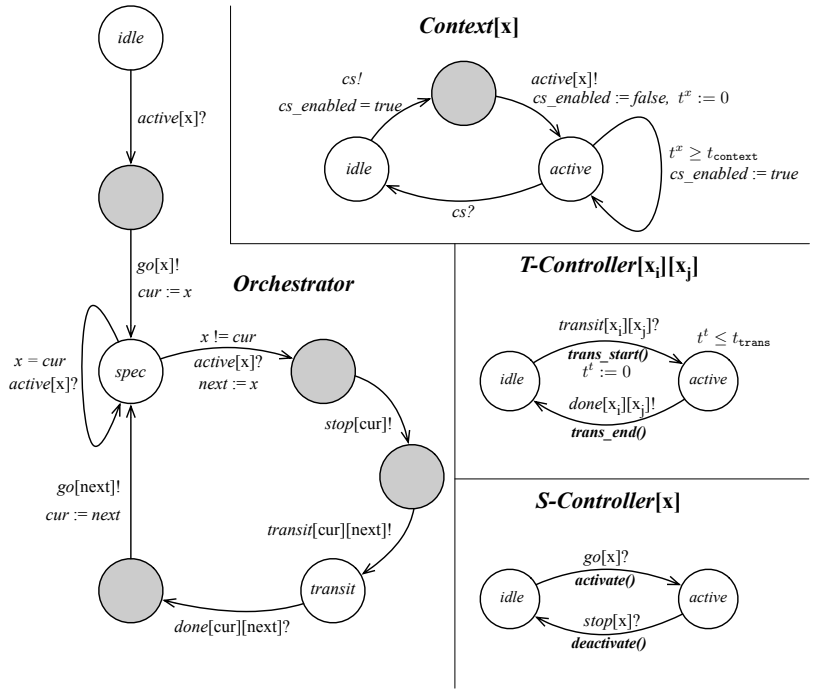


Figure 7.7: Timed automata that modeling the interactions among the orchestrator, contexts and controllers. Gray states represents *committed* locations.

Synchronization channels The coordination among automata happens via the synchronization of the transitions of different automata on binary synchronization channels. An edge labeled with *channel*! synchronizes with another labeled *channel*?. Given n contexts, we have defined the following synchronization channels:

— n *active* channels, one channel per each context, each channel gets fired when a context becomes active.

— n *go* and n *stop* channels, to activate and deactivate respectively the specification controllers.

— $(n^2 - n)$ *transit* and $(n^2 - n)$ *done* channels, respectively to activate each transition controller and get notified with it has terminated its execution.

— *cs* is one *broadcast channel* where one sender (*cs*!) can synchronize with an arbitrary number of receivers (*cs*?). It is used to notify all contexts of a context switch.

States Committed locations (represented in gray in Figure 7.7 are states where time delay is not allowed and the only possible transition is the one outgoing the committed location. Thus, all actions from/to a committed state are considered atomic.

Initially we have that all the automata are in the *idle* state. At any other given moment, context, transition and specification controllers can either be in

the *active* or the *idle* states while the orchestrator can be in the *spec* or *transit* states.

Local clock variables Contexts and transition controllers have a local clock variable, i.e. t^x and t^t respectively. The context use t^x to measure the *minimum* time to be active. The transition controller use t^t to bound the *maximum* time that they can be active. Both clock variables are reset every time the automata move to their *active* states.

Synchronization with the Mealy machines The *trans_start()*, *trans_end()*, *activate()* and *deactivate()* functions are executed every time a transition is taken from *idle* to *active* in a transition or specification controller. These functions are responsible for managing the *activation inputs* in $\mathcal{I}^{||}$ and consequently affecting the behavior of $\Lambda^{||}$. Specifically for any specification controller $S[i]$ and transition controller $T[i][j]$:

— *activate*() sets to true the activation input of the Mealy machine λ'_i, which is the only machine related to the specification controller $S[i]$. It also sets the output $Active \in \mathcal{O}$ to true.

— *activate*() sets to false the activation input of the Mealy machine λ'_i. It also sets *Active* to false.

— *trans_start*() sets to true the activation input of a Mealy machine $k'_{r_i r_j} \in K'$. Which Mealy machine to activate depends on the current location of the robot r_i and its destination r_j.

— *trans_end*() sets to true the activation input of $k'_{r_i r_j} \in K'$ to false.

Remark. $Active \in \mathcal{O}$ is *true* if and only if any specification controller is active

We want to prove that our orchestration system satisfies the following properties:

— Initially all automata wait for a context to become active.

— After the first context became active there is exactly one context active at all time.

— Any context stays active for at least $t_{\texttt{context}}$ time units.

— No context can become active two times consecutively.

— No context can become active two times consecutively.

— After the first context has become active, there is always one specification controller or one transition controller active but they are never active together.

— Transition controllers can be active at most $t_{\texttt{trans}}$ time units.

— Every time a context becomes active after at most $t_{\texttt{trans}}$ time units the specification controller related to the active context also becomes active.

. To do so, we show in what follows a series of Lemmas used in the final proof.

In the following we assume that all automaton before the execution are in the *idle* state.

Lemma 7.5.1. Under the initial condition no automaton can take a transition except for a context, and exactly one context can transition to the *active* state.

Proof. We will prove this by induction on the number of contexts. Given n context, we have that:

$n = 1$ In this case we only have one context $C[0]$, one orchestrator O and one specification controller $S[0]$. We have that O and $S[0]$ are waiting on the synchronization channels *active* and *go* respectively. *cs_enabled* is initially set to *true*, so $C[0]$ is free to move from *idle* to *active*. The atomic transition from idle to active is synchronized on the *cs* and *active* channels. It first sends on the broadcast channel *cs*! which is non-blocking and it has no effects since there is no other context enabled to receive on *cs*?. In the same transition, it synchronizes with the orchestrator via the *active*[0] channel and the orchestrator can now synchronize with $S[0]$ via the $go[x]$ channel. We conclude that initially, no other automaton could have moved from the *idle* state but the context automaton $C[0]$.

$n = 2$ In this case we have two contexts $C[0]$ and $C[1]$, one orchestrator O, two specification controllers $S[0]$ and $S[1]$ and two transition controllers $T[0][1]$ and $T[1][0]$. Similar to the case of $n = 1$, the orchestrator, the specification controllers and the transition controllers are all waiting on the synchronization channels *active*, *go* and *transit* respectively. The only automata that can transition are $C[0]$ and $C[1]$. Both are enabled with a transition from *idle* to *active*. The first context takes the transition set *cs_enabled* to false which forces the other context to stay in the *idle* state. We conclude that also for $n = 2$ we have that only one context can take the first transition.

Induction hypothesis $n = k$ Let us assume that our lemma holds for a number of context $n = k$.

Induction step $n = k+1$ We will prove that the lemma holds for $n = k+1$. By adding an additional context we have that all contexts $C[0], C[1], ..., C[k]$ are enabled to transition from the *idle* to the *active*. By our induction hypothesis we have that only one context, let's say context $C[i]$ $(0 \leq i \leq k)$, can effectively take the transition from *idle* to *active*, and by the base case (with $n = 2$), as soon as $C[i]$ takes such transition then we know that $C[k + 1]$ cannot do it.

We conclude that for any number of contexts n, one context is always the first automaton to move from *idle* to *active* state while the others remain in the *idle* state. □

Lemma 7.5.2. For any number n of contexts, there is exactly one context in the *active* state at any moment in time after the first context has become active. Any context stays in the *active* state for at least $t_{\texttt{context}}$ time units.

Proof. According to Lemma 7.5.1 contexts are the first automata to move. Initially, any instance can take the outgoing transition from the *idle* state. Non-deterministically, one of the instances will set *cs_enabled* to false, and due to the atomicity guaranteed by committed states, all the other context instances will remain in the *idle* state. The first context instance to take the transition sets *cs_enabled* to false before reaching the *active* state so that no other context can become active. After $t_{\texttt{context}}$ time units the active context sets *cs_enabled* to true so that another context is enabled to become active. As soon as another context exits from the *idle* state it synchronizes with the currently active context on the *cs* channel so that the active context transitions to the *idle* state. □

Lemma 7.5.3. No context can be activated twice in a row.

Proof. The only way for an active context $C[i]$ to move to the idle state is to synchronize via *cs*? channel, which can only be done when a context $C[j]$ where $i \neq j$ takes the transition from idle to active; thus we have that context $C[i]$ must wait for at least one turn before proceeding. □

The orchestrator keeps track of the current active context *cur* and as soon as another context instance x becomes active:

1. it stops the execution of the specification controller currently active, i.e. $S[cur]$.
2. it activates transition controller from context *cur* to context x, i.e. $T[cur][x]$.
3. it waits for transition controller to terminate its execution waiting for the $done[cur][x]$ synchronization channel to fire.
4. it activates the specification controller of context x, i.e. $S[x]$.

Lemma 7.5.4. After the first context becomes active, there is always a specification controller or a transition controller in the *active* state, but never both active together.

Proof. Operations 1), 2) and 4) are *atomic operations* and they are modeled as *committed locations* in Figure 7.7. The first time a context becomes active the orchestrator activates a specification controller in the same time step. During any context switch, at the same time instance, the orchestrator stops the current specification controller which transitions to the *idle* state and activates a transition controller which transits to the *active* state. When the transition controller terminates its execution and transits to the *idle* state, it synchronizes with the orchestrator that activates the next specification controller in the same atomic operation. □

Lemma 7.5.5. A transition controller can be active at most $t_{\texttt{trans}}$ time units.

Proof. Because of the invariant in the *active* state of the transition controller, the automata have to take the transition to the idle state when $t^t \leq t_{\texttt{trans}}$. □

Theorem 7.5.6. Given n contexts, n specification controllers, n^2-n transition controllers and assuming that $t_{\texttt{trans}} < t_{\texttt{context}}$, we have that:

1. Every time a context $C[i]$ is active, then $S[i]$ becomes active after at most $t_{\texttt{trans}}$ steps and no transition or specification controller beside $S[i]$ can become active.

Proof. Initial phase If the orchestrator and controllers are all in the *idle* state, according to Lemma 7.5.1 and Lemma 7.5.4 the system is in the initial step waiting for one (and only one) context to become active. If $C[i]$ becomes active, then the orchestrator immediately activates $S[i]$ by synchronizing on the *go* channel.
Context Switch. Assume that we are in a condition where a context $C[j]$ is currently active (where $j \neq i$), then according to Lemma 7.5.2 $C[i]$ cannot become active before $t_{\texttt{context}}$ time units. Assume that $C[i]$ becomes active $t_{\texttt{context}}$ units after $C[i]$ became active, then according to Lemma 7.5.5 and Lemma 7.5.4 and our assumption $t_{\texttt{trans}} < t_{\texttt{context}}$, we have that $S[j]$ must be active when $C[i]$ becomes active. The orchestrator will then stop $S[j]$ and activate $T[j][i]$. After at most $t_{\texttt{trans}}$ time units $S[i]$ will become active. Because we assumed that $t_{\texttt{trans}} < t_{\texttt{context}}$ and we proved that 1) another context $C[j] \neq C[i]$ cannot become active before $t_{\texttt{context}}$ time units (Lemma 7.5.2) and that 2) $C[i]$ cannot become active twice in the row (Lemma 7.5.3), we conclude that $S[i]$ must become active.

□

Starvation Our orchestration system only assumes that the context are mutually exclusive and that they stay active for at least $t_{\texttt{context}}$ time units. We have no assumptions on how the contexts are scheduled. It can happen that some contexts are never becoming active and consequently the corresponding specification controller will never be activated by the orchestrator.

Remark. Assuming a fair scheduler that will activates all the context infinitely often and that $t_{\texttt{trans}} < t_{\texttt{context}}$ then according to Theorem 7.5.6 we can also claim that all the specification controllers will be active infinitely often, and no context (specification controlloer) would starve.

Model checking We modeled the automata in UPPAAL[2] [224] and successfully verified our model against the following properties.

'There is never more than one T-controller active at any given time'

```
A[] forall (i: N) forall (j: N) forall (l: N) forall (m: N)
T_controller(i,l).Active & T_controller(j,m).Active imply (i==j &
                                l==m)
```

'There is always at most one context active at any given time'

```
A[] forall (i: N) forall (j: N) Context(i).Active &
                Context(j).Active imply (i==j)
```

[2]UPPAAL model available:
`rebrand.ly/cromeuppaalmodel`

'There is always at most one S-controller active at any given time'

```
A[] forall (i: N) forall (j: N) S-controller(i).Active &
          S-controller(j).Active imply (i==j)
```

'If any context becomes active its S-controller will eventually become active'

```
Context(i).Active --> S_controller(i).Active
```

'There are no deadlocks'

```
A[] not deadlock
```

7.5.3 Time Synchronization

The timed-automata model the *dynamic behavior* of the system and have a precise notion of *time* [224], the Mealy machine determines what the robot should do when it reacts, however, there are no constraints on *when* the reaction should happen. Let us define when the Mealy machine reactions happen and how they synchronize them with the timed automata. The timed automata modeled in Uppaal uses a *continuous time model*. We have local clocks to every state machine that determines when a transition should fire and synchronize with another transition if a synchronization channel is present. We discretize the time in fixed time-units t. At each t, a transition can be taken by any timed automata (e.g. if the transition guards allow it). In parallel with the timed automata, at each t a transition must happen in every Mealy machine participating in the composition $\Lambda^{||}$, that is at every t each Mealy machine evaluates its inputs and set its outputs.

Theorem 7.5.7. [Mission Satisfaction] Given a contextual mission $\mathcal{M} = (X, \Gamma, AC, CT)$, and assuming that:

1. for every specification in γ there is a specification controller λ that realize it.
2. there is a transition controller k for every possible context switch and where $t_{\texttt{trans}} < t_{\texttt{context}}$.

Then the behaviors produced by $\Lambda^{||}$ together with the orchestration system of CROME will always satisfy the contextual mission as defined in (7.4.1).

Proof. Theorem 7.5.6 guarantees that for any context becoming active the orchestration system guarantees that a specification controller will become active after at most $t_{\texttt{trans}}$ time units. The behavior of the specification and transition controllers automata is linked to the *actual* (i.e., the Mealy machines) specification and transition controllers via the execution of the *activate()*, *deactivate()*, *trans_start()* and *trans_end()* functions.

We have that for any context $x_i \in X$ becoming active at any time $t \in \mathbb{N}$ where $CT(ACT(t)) = \gamma_i$ and $\lambda_i \models \gamma_i$, λ_i' will be *active* in at most $t_{\texttt{trans}}$ time units. According to the remark in Section 7.5.1 the behavior produced by λ_i' is equivalent to λ_i when λ_i' is active and according to the remark in Section 7.5.2 *Active* is true if an only if a specification controller is active. Because λ_i satisfies γ_i by construction, we have that the mission $\mathcal{M}$ is always satisfied by the robot $\mathcal{R}$ according to the definition of satisfaction defined in (7.4.1).

□

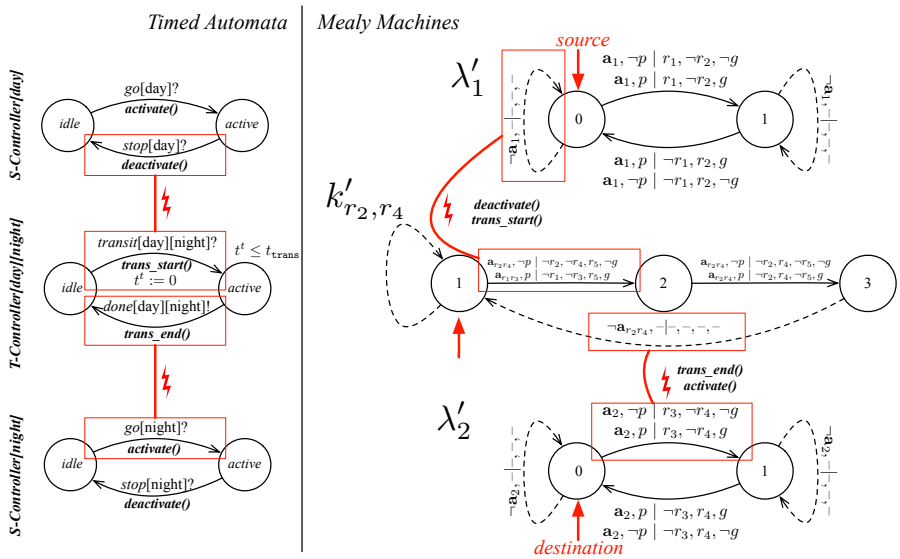


Figure 7.8: Synchronization between Timed Automata and Mealy Machines. The red arrows indicate the state of the machines at time t, when we have a context switch from context *day* to *night*. The red boxes connected indicate transitions that happen in parallel.

7.6 Orchestration in the running example

Let us consider the two contexts in our example: *day* and *night*. We have two instances of *S-Controllers*, i.e. *S[day]* and *S[night]* and two specification controllers λ'_1 and λ'_2 realized from the specifications $CT(AC(day)) = \gamma_1$ and $CT(AC(night)) = \gamma_2$ respectively. In order to show simple controllers we will only consider the specification of the three goals $\mathcal{G}_d$, $\mathcal{G}_n$ and $\mathcal{G}_g$, leaving out $\mathcal{G}_r$ i.e. where the robot has to register a person in the *next* step. We will consider the full running example in Section 7.6.1. Figure 7.8 shows the two Mealy machines (λ'_1 and λ'_2) realizing γ_1 (7.11) and γ_2 (7.12) which have been modified so that they accept the *activation inputs* $\mathbf{a}_1$ and $\mathbf{a}_2$. For presentation purposes in Figure 7.8 we use the letters $p, r_1, r_2, r_3, r_4, r_5$ and g to represent the atomic propositions *person*, r_1, r_2, r_3, r_4, r_5 and *greet*, respectively.

At any point in time only one of the *activation inputs* of $\Lambda^{||}$ is set to *true*, hence only one of the Mealy machines composing $\Lambda^{||}$ is active. The functions that are executing on the transitions of the *S-Controllers* and *T-Controllers* are managed by the activation inputs. Specifically:

— **activate()** is a function executed by any *S-Controller* template instance, e.g. $S[day]$ and $S[night]$ in our example. Its execution is triggered by an *S-Controller* transitioning from *idle* to *active* state and produces the activation of a specification controller λ'. Specifically, any *S-Controller* instantiated for a context x_i that transition at time t (i.e. $AC(t) = x_i$) from *idle* to *active* will activate the specification controller λ'_i, which realizes the specification $\gamma_i = CT(x_i)$. In Figure 7.8, **activate()** of *S[day]* will set the activation input of λ'_1 (i.e. $\mathbf{a}_1$) to *true*, while **activate()** of

S[night] will set the activation input of λ'_2 (i.e. $\mathbf{a}_2$) to *true*.

— **deactivate()** is the opposite of the **activate()** function. It sets the activation inputs of the corresponding specification controllers to *false*. The function gets triggered after a context-switch when the *Orchestrator* synchronizes with the active *S-Controller* that transition from the *active* to *idle* state.

— **trans_start()** for any contexts switch, e.g. from *day* to *night*, it activates one transition controller. However, unlike the specification controller, there could be more than one transition controller for each instance of the *T-Controller* template. Each transition controller is responsible for guiding the robot from one location to another. The transition controller that needs to be activated after a context-switch depends on *where* is the robot when the context-switch happens and where it needs to be under the new context. In the example in Figure 7.8, the control has to pass from λ'_1 to λ'_2, the transition controller that needs to be activated depends on the current state of λ'_1 to λ'_2 when the context-switch happens.

— **trans_end()** gets executed when the transition controller has reached its final state and it sets its activation input to *false*.

In the example in Figure 7.8, let us assume that we have a context-switch at time t where we have that λ'_1 is active and is in the state 0 and λ'_2 is inactive and is in the state 0. Each state corresponds to the last region that has been reached by the robot, that is region r_2 for λ'_1 and region r_4 for λ'_2.

At time t S[day] transitions from *active* to *idle* and *T[day][night]* transitions from *idle* to *active*. **deactivate()** set $\mathbf{a}_1$ to *false*. **trans_start()** chooses the transition controller that can connect the state 0 of λ'_1 to the state 0 of λ'_2 indicated as *source* and *destination* in Figure 7.8. In our example it sets $\mathbf{a}_{r_3r_4}$ to *true*.

At time $t+1$ *S[day]* and *S[night]* are in the *idle* state, and *T[day][night]* is in the *active* state. The transition controller $k'_{r_2r_4}$ moves from state 2 to state 3.

At time $t+2$ The transition controller $k'_{r_2r_4}$ has reached its final state and *T[day][night]* transition from *active* to *idle* causing **trans_end()** to execute which sets $\mathbf{a}_{r_2r_4}$ to *false*. In the same instance, *S[night]* transitions from *idle* to *active* while **activate()** sets $\mathbf{a}_2$ to *true*.

7.6.1 Timeline of the full running example

Let us now consider the full running example where we have all four goals $\mathcal{G}_d$, $\mathcal{G}_n$, $\mathcal{G}_g$ and $\mathcal{G}_r$, composed in the CGG under the two scenarios: *1)* $\mathcal{G}_{dgr}$ where the robot greets and registers peoples during the day while patrolling locations r_1 and r_2 and *2)* $\mathcal{G}_{ng}$ where the robot only greets people during the night while patrolling locations r_3 and r_4.

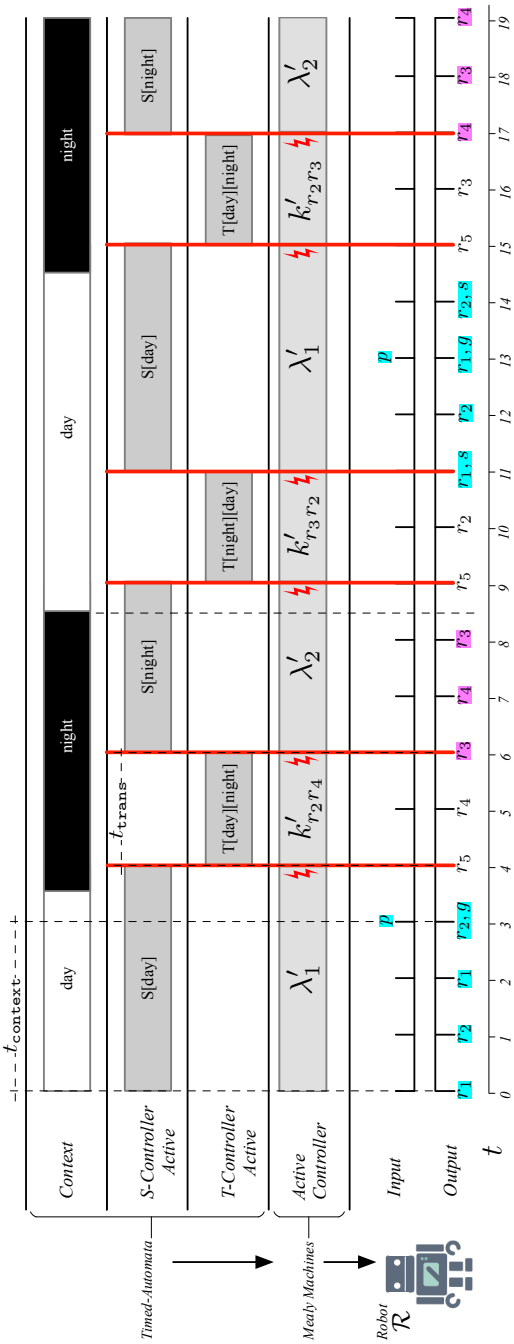


Figure 7.9: Timeline of the running example.

Figure 7.9 shows the timeline of events where the contexts switches from *day* to *night* after a minimum context time of t_{context} time units. The timed automaton synchronizes after every context-switch and the *T-Controllers* are active for at most t_{trans} time units. At the bottom of the figure we can see the

trace of the robot $\mathcal{R}$ defined by its *input-output* relationship where s represents the controllable atomic proposition *register*.

The output traces highlighted in blue are the trace projections for the specification γ_1 (i.e. related to the goal $\mathcal{G}_{dgr}$) while the outputs highlighted in pink are those for the specification γ_2 (i.e. related to the goal $\mathcal{G}_{ng}$). Note that at time-step 3 the robot perceives a person (p) and while it greets immediately (g), the registration (s) will be executed at step 11. That is the next time the context *day* is active *and* the robot has finished the transition period.

7.7 Mission satisfaction and CGG satisfaction Relationship

In Theorem 7.5.7 we proved that the robot behavior produced by CROME satisfies the contextual mission. However, such behavior could *not* satisfy the mission as specified in the root of the CGG. This is not a problem as we never required that, however, we could still be interested in having in the root node of the CGG a specification that represents an *abstraction* of the behavior generated by the orchestration system.

Let $\mathcal{C} = (\psi, \phi)$ be the contract formalizing the root goal of the CGG $\mathcal{G}$ which is produced by the conjunction of all the mission scenarios. Each mission scenarios is built from specifications that do not contain the notion of context while the final robot behavior is produced by the orchestration of different controllers according to *1)* the context and *2)* the transition of the robot from one context to another. It can happen that, according to the robotic pattern chosen by the designer to model the different goals, the robot behavior while satisfying the mission does not satisfy the specification in the CGG.

For example, in our running example we have the goal $\mathcal{G}_r$ formalized by the trigger pattern *BoundedReaction* which performs the counteraction (i.e. *register*) in the *next* time instant, every time and only when the pre-condition (i.e. *person*) is true. This goal is expressed in LTL as

$$\mathbf{G}(person \leftrightarrow \mathbf{X} register) \tag{7.55}$$

However, if we consider the example in Figure 7.9, since a person is detected at $t = 3$ in order to satisfy (7.55) $\mathcal{R}$ would need to set *register* to true when $t = 4$, but since the context changes at $t = 4$ it will satisfy the postcondition only the *next* time when the *day* context is true and $\mathcal{R}$ is not transitioning, i.e. $t = 11$.

Such problems only happen with the *'avoindance'* and *'trigger'* patterns of the robotic catalog [225]. The *'core movement'* patterns (e.g. *patrolling* a set of locations $\mathbf{G}(\mathbf{F}(r_1, r_2, ...r_n))$ or *'visit'* patterns (e.g. $\mathbf{F}(r_1, r_2, ...r_n)$) are not affected by the context-switch since they only indicate the order in which location specified must be visited; they do not forbid to visit other locations. However we have to assume that the set of regions refer by each mission scenario is disjoint.

Definition 7.7.1 (Disjoint scenarios)**.** Let $\mathcal{AP}_r^i \subseteq \mathcal{AP}_r$ be the subset of atomic propositions referring to region variables in the specification $\gamma_i \in \Gamma$. Then for any $\gamma_i, \gamma_j \in \Gamma$ where $i \neq j$ if $\mathcal{AP}_r^i \cap \mathcal{AP}_r^j = \emptyset$ then we say that we have *disjoint scenarios*.

Remark. The orchestration system of CROME will take care of ordering the location to be visited according to the context while satisfying the original pattern formulation. Since each context is mutually exclusive assuming that we have disjoint scenarios (7.7.1) as long as the visit or patrolling order is kept within the scenarios then any trace of $\mathcal{R}$ also satisfies the conjunction of the mission scenarios, i.e. the root node of the CGG.

To have a specification that always satisfies the traces produced by CROME and assuming disjoint scenarios, we have modified the following robotic patterns as follows. We define a boolean variable $c_i = x_i \wedge \mathit{Active}$ for any context $x_i \in X$ and where $\mathit{Active} \in \mathcal{O}$ is the active Boolean signal of $\mathcal{R}$ determining when the robot is *not* transitioning from one specification to another. In the following, we show the modified patterns together with their original version:

$$\begin{aligned} \textit{Instant Reaction} \quad & \mathbf{G}(p_1 \rightarrow p_2) \\ \text{modified with} \quad & \mathbf{G}((p_1 \wedge c_i) \rightarrow (p_2 \wedge c_i)) \end{aligned}$$

$$\begin{aligned} \textit{Delayed Reaction} \quad & \mathbf{G}(p_1 \rightarrow \mathbf{F}(p_2)) \\ \text{modified with} \quad & \mathbf{G}((p_1 \wedge c_i) \rightarrow \mathbf{F}(p_2 \wedge c_i)) \end{aligned}$$

$$\begin{aligned} \textit{Prompt Reaction} \quad & \mathbf{G}(p_1 \rightarrow \mathbf{X}(p_2)) \\ \text{modified with} \quad & \mathbf{G}((p_1 \wedge c_i) \rightarrow \mathbf{X}(\neg c_i \mathbf{W}(p_2 \wedge c_i))) \end{aligned}$$

$$\begin{aligned} \textit{Bound Reaction} \quad & \mathbf{G}(p_1 \leftrightarrow p_2) \\ \text{modified with} \quad & \mathbf{G}((p_1 \wedge c_i) \leftrightarrow (p_2 \wedge c_i)) \end{aligned}$$

$$\begin{aligned} \textit{Bound Delay} \quad & \mathbf{G}(p_1 \leftrightarrow \mathbf{F}(p_2)) \\ \text{modified with} \quad & \mathbf{G}((p_1 \wedge c_i) \leftrightarrow \mathbf{X}(\neg c_i \mathbf{W}(p_2 \wedge c_i))) \end{aligned}$$

$$\begin{aligned} \textit{Wait} \quad & r_1 \mathbf{U} p_3 \\ \text{modified with} \quad & ((r_1 \wedge c_i) \vee \neg c_i) \mathbf{U} (p_1 \wedge c_i) \end{aligned}$$

Where $p_1 \in \{\Omega_S \cup \Omega_R \cup \Omega_A\}$, $p_2 \in \{\Omega_R \cup \Omega_A\}$, $p_2 \in \{\Omega_S \cup \Omega_A\}$ and $r_1 \in \Omega_R$.

Remark. Realizing the specification directly with the modified patterns does not produce a controller that satisfies the mission, since the modifications were only applied to some of the patterns and the behavior of *Active* variable is not always specified.

7.8 Evaluation

CROME is implemented as a tool and it is available open source [3]. The tool allows the designer to:

1. model the mission environment
2. model the goals of the mission using robotic patterns
3. build the CGG and analyze consistency and completeness
4. automatically realize all specification and transition controllers
5. run the system orchestration of CROME
6. generate random simulations of the robotic mission

The tool uses NuXMV [177] and STRIX [226] as back-end engines to check respectively the satisfiability and realizability of the specifications.

By using CROME the designer produces specifications in an incremental and modular way. Goals are specified in terms of contexts and patterns and CROME automatically builds the CGG and, thanks to the orchestration system, it can produce controllers that satisfy a contextual mission. In the evaluation, using the tool, we want to compare the performance of CROME in the synthesis and orchestration of modular controllers with respect to the synthesis of a monolithic LTL specification which can be realized in a controller that produces the same traces produced by CROME (where it is relatively easy to express one).

Due to the difficulties of using LTL to express contextual missions presented in Section 7.4.2 we could only produce a monolithic LTL specification for reasonably small examples. Specifically, we have approximated the running example presented in this paper as a monolithic LTL formula. In the approximation, we have fixed a minimal duration of the contexts *day* and *night*. This minimal duration is set long enough to allow the synthesis of the transition between the different controllers that are active in the different contexts. Furthermore, as the example includes only two locations for each context the global safety conditions on the different locations (i.e. the condition specifying the adjacent locations that the robot can visit) is enough to ensure that local safety conditions are maintained under each context separately. This is not generally the case as we would have to include specialized *per context* location safety conditions that would ensure continuity of the controller within each context. That is, we have would have to explicitly model that the robot has to come back to the same location after it has left due to a context switch, as explained in the *Problem 4* of Section 7.4.2.

To express the contextual mission in LTL we have to embed the notion context and active signal also in all the specifications. The resulting controllers for the transitions between contexts are not as tight as the optimal controllers included in CROME. However as our running example is relatively simple, we can give further specifications that narrow down the behavior of the transition controllers to emulate the behavior produced by CROME. Namely, we force

[3] CROME tool: `rebrand.ly/crometool`

the transition controller to immediately go from wherever it is to r_5 and move on from r_5 immediately as we want the transition to take an as short time as possible (please check the map in Figure 7.1). Note that, without access to the transition controllers and the traces produced CROME we would not be able to give such a tight specification leaving the synthesizer much more freedom on how to synthesize the transition controllers.

We have ultimately produced two specifications. The first one only embeds the notion of context and active signal in the specification. The traces produced by CROME always satisfy such a specification, however, the opposite is not true. That is, is not always the case that the controller produce but such specification satisfy the mission due to our stringent transition controllers requirements. In the second specification, after looking at the transition controller produced by CROME, we have *forced* the removal of some behaviors that would violate the mission, i.e. refining the specification. The full monolithic LTL specifications and the test conducted can be found online[4]. In the rest of the paper, we will only consider the LTL specification that always satisfies the contextual mission, the results are very similar for the other, more *relaxed*, specification.

	Monolithic LTL	CROME
Synthesis Time (sec)	35.6	0.14
Number of States	119	5 and 3
Number of Transitions	357	10 and 6

Table 7.1: Comparison of CROME with the synthesis from a single LTL formula in order to realize the running example.

Table 7.1 shows the synthesis time and the size of the controller produced by the two approaches, monolithic LTL vs CROME. For the synthesis time, in the monolithic LTL, we had to add extra variables to keep track of the context, active signal, and transition conditions from one context to another remarkably increasing the size of the LTL formula that took 35.6 seconds to synthesize, producing a controller having 119 and 357 transitions. On the other hand CROME, thanks to the lean and modular approach only had to synthesize the controllers for the two mutually exclusive contexts: *day* and *night*. Specifically, it had to synthesize the specification γ_1 (7.47) related to the goal $\mathcal{G}_{ng}$ and the specification γ_2 (7.48) related to the goal $\mathcal{G}_{dgr}$. These specifications are much smaller and it took only 0.14 seconds to synthesize both of them, 254 timesfaster than the monolithic synthesis. The controller produced for the two scenarios consists of only 5 and 3 states and 10 and 6 , respectively.

However, in our evaluation so far we did not take into consideration the *transition controllers*. Each transition controller has the objective to guide the robot from one location to another. There are two principal factors to take into consideration when realizing the transition controller so that the contextual mission can always be satisfied:

1. *Number of pairs of locations.* According to the mission specification, there could be several contexts switches some of which may or may not

[4]CROME evaluation: `rebrand.ly/cromeevaluation`

require the robot to change location. Let us assume the worst possible case where all context switches require the robot to move to a different set of locations. Let $\Omega_R = \{\Omega_{R1} \cup \Omega_{R2} \cup, ..., \cup \Omega_{Rn}\}$ be the set of regions where $\Omega_{R1}, \Omega_{R2}, ..., \Omega_{Rn}$ are disjoint sets of regions, where each set is related to a scenario activated in a mutually exclusive context $x_1, x_2, ..., x_n$. Then, we have that the number of pairs of locations for which we need a transition controller is:

$$(|\Omega_{R1}| \times |\Omega_{R2}| \times ... \times |\Omega_{Rn}|) \times 2 \tag{7.56}$$

since we need to consider all possible pairs of locations and their permutation.

2. *Transition controller generation.* Given the map of the environment, generating a transition controller is essentially solving a '*shortest path problem*'. We do not have to use reactive synthesis to produce the transition controller, we can use any algorithm that solves the shortest path problem (e.g. Dijkstra).

For example, in our running example, there could be at most 8 transition controllers needed if we consider the formula in (7.56). If we want to include also *adjacent locations* (i.e. r_5 in our example) then we could have at most 16 transition controllers. The time took by CROME to try to realize all 16 of them via reactive synthesis is 0.92 seconds. This was performed by realizing a specification that starting from the *source* location guarantees reaching the *destination* location in N steps, where $N \leq t_{\texttt{trans}}$. Our algorithm tries all $1 < N < t_{\texttt{trans}}$ until it realizes a controller (if exists). The algorithm generating transition controllers can be improved by, for example, taking into consideration the adjacencies information of the environment and not synthesizing most of the controllers. Let us consider the environment of our running example depicted in Figure 7.1, an improved algorithm for transition controller would only produce 1 controller (i.e. the controller going to location r_5) for any context switch in any location since for any context switch is enough for the robot to go to r_5 to be able to reach any location.

Furthermore, we have to consider that the final specifications γ_1 and γ_2 are not given *as is* by the designer, instead, they are produced by CROME. To produce γ_1 and γ_2 from a set of the four input goals inserted by the designer, CROME has performed 11 satisfiability checks and 10 validity checks and has ultimately produced the CGG which took a total of 0.54 seconds.

Finally, if we take into consideration the time needed to produce the CGG and all 16 transition controllers we would have a total synthesis time of 1.60 seconds, which is still a 22 timesimprovement compared to synthesizing the monolithic LTL specification.

7.9 Conclusions

In this paper, we introduced CROME, a design framework for capturing and formalizing robotic mission requirements. CROME facilitates the translation of informal requirements in terms of goals by leveraging on specification patterns

and the notion of *context*. It then formalizes the goals in terms of assume-guarantee contracts and uses an improved version of the contract-based goal graph (CGG) introduced in our previous work [200] to analyze the mission specification and detect inconsistencies.

We have tackled the problem of dynamically switching controllers for different missions scenarios, where each scenario is activated in one mutually exclusive context. We have first formally defined what does it mean to satisfy a contextual mission and we have then presented an approach that orchestrates the different controllers in a way that the mission is always satisfied. We have then proved that our approach works for any number of mutually exclusive contexts. Finally, we have shown how our approach works on a running example and evaluate it against a monolithic specification written in LTL. Our results show that using CROME to synthesize contextual mission controller produces a much more lean and modular specification which greatly improves the synthesis time. In the future, we plan to extend our approach by formally defining a new logic that can express the class of problems described in this paper. Finally, we plan to validate our approach on a larger industrial case study.

Acknowledgments

This work was supported by the Wallenberg Al Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation

Bibliography

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[2] B. H. Cheng, H. Giese, P. Inverardi, J. Magee, R. de Lemos, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research road map," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[3] R. de Lemos et al., "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–32.

[4] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops." *Software engineering for self-adaptive systems*, vol. 5525, pp. 48–70, 2009.

[5] J. R. Boyd, "The essence of winning and losing," *Unpublished lecture notes*, 1996.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell, "Toward an architecture for never-ending language learning." in *AAAI*, vol. 5, 2010, p. 3.

[8] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2017.

[9] D. Schneider and M. Trapp, "Conditional safety certification of open adaptive systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 2, pp. 1–20, 2013.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[11] B. H. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe *et al.*, "Using models at runtime to address assurance for self-adaptive systems," in *Models@ run. time*. Springer, 2014, pp. 101–136.

[12] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Transactions on Software Engineering*, 2017.

[13] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone *et al.*, "Contracts for system design," *Foundations and Trends in Electronic Design Automation*, vol. 12, no. 2-3, pp. 124–400, 2018.

[14] P. Nuzzo, A. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proc. IEEE*, vol. 103, no. 11, Nov. 2015.

[15] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[16] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[17] A. Church, "Logic, arithmetic, and automata," *Journal of Symbolic Logic*, vol. 29, no. 4, 1964.

[18] M. O. Rabin, *Automata on infinite objects and Church's problem*. American Mathematical Soc., 1972, vol. 13.

[19] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive (1) designs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 364–380.

[20] K. Havelund and G. Roşu, "Runtime verification," in *Computer Aided Verification (CAV'01) satellite workshop*, ser. ENTCS, vol. 55, 2001.

[21] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[22] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "Lola: Runtime monitoring of synchronous systems," in *TIME'05*. IEEE Computer Society Press, June 2005, pp. 166–174.

[23] M. Jackson, "The world and the machine," in *1995 17th International Conference on Software Engineering*. IEEE, 1995, pp. 283–283.

[24] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of computer programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[25] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. 203–236, 2004.

[26] E. S. Yu, "Towards modelling and reasoning support for early-phase requirements engineering," in *Proceedings of ISRE'97: 3rd IEEE International Symposium on Requirements Engineering.* IEEE, 1997, pp. 226–235.

[27] J. Horkoff, F. B. Aydemir, E. Cardoso, T. Li, A. Maté, E. Paja, M. Salnitri, L. Piras, J. Mylopoulos, and P. Giorgini, "Goal-oriented requirements engineering: an extended systematic mapping study," *Requirements engineering*, vol. 24, no. 2, pp. 133–160, 2019.

[28] A. Van Lamsweerde, *Requirements engineering: From system goals to UML models to software.* Chichester, UK: John Wiley & Sons, 2009, vol. 10.

[29] A. Van Lamsweerde, R. Darimont, and P. Massonet, "Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt," in *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95).* IEEE, 1995, pp. 194–203.

[30] A. Van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 5–19.

[31] D. Zowghi and V. Gervasi, "On the interplay between consistency, completeness, and correctness in requirements evolution," *Information and Software Technology*, vol. 45, no. 14, pp. 993–1009, 2003.

[32] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Foundations of Software Engineering (FSE).* ACM, 2015.

[33] M. Guo and D. V. Dimarogonas, "Multi-agent plan reconfiguration under local LTL specifications," *The International Journal of Robotics Research*, 2015.

[34] C. Finucane, G. Jing, and H. Kress-Gazit, "LTLMoP: Experimenting with language, temporal logic and robot control," in *International Conference on Intelligent Robots and Systems (IROS).* IEEE, 2010, pp. 1988–1993.

[35] C. Menghi, S. Garcia, P. Pelliccione, and J. Tumova, "Multi-robot LTL Planning Under Uncertainty," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 399–417.

[36] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, "Optimal multi-robot path planning with Temporal Logic constraints," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011.* IEEE, 2011.

[37] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, no. 2, pp. 343–352, 2009.

[38] M. Guo, K. H. Johansson, and D. V. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2013.

[39] E. M. Wolff, U. Topcu, and R. M. Murray, "Automaton-Guided Controller Synthesis for Nonlinear Systems with Temporal Logic," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013.

[40] H. Kress-Gazit, "Robot challenges: Toward development of verication and synthesis techniques [errata]," *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, pp. 108–109, 2011.

[41] S. Maoz and J. O. Ringert, "Synthesizing a Lego Forklift Controller in GR(1): A Case Study," in *Proceedings Fourth Workshop on Synthesis (SYNT)*, 2015.

[42] S. Maoz and Y. Sa'ar, "AspectLTL: an aspect language for LTL specifications," in *International conference on Aspect-oriented software development*. ACM, 2011.

[43] S. Maoz and J. O. Ringert, "On well-separation of GR(1) specifications," in *Foundations of Software Engineering (FSE)*. ACM, 2016.

[44] Y. Shoukry, P. Nuzzo, A. Balkan, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming," in *Proc. Int. Conf. Decision and Control*, Dec. 2017.

[45] X. Sun, R. Nambiar, M. Melhorn, Y. Shoukry, and P. Nuzzo, "DoS-resilient multi-robot temporal logic motion planning," in *Proc. International Conference on Robotics and Automation (ICRA)*, 2019, pp. 6051–6057.

[46] G. J. Holzmann, "The logic of bugs," in *Foundations of Software Engineering (FSE)*. ACM, 2002.

[47] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: An automated approach," *Automated Software Engg.*, vol. 14, no. 3, 2007.

[48] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification Patterns for Robotic Missions," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[49] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 411–420.

[50] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proc. of ICSE'05*. ACM, 2005, pp. 372–381.

[51] L. Grunske, "Specification patterns for probabilistic quality properties," in *30th International Conference on Software Engineering (ICSE08)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM Press, 2008, pp. 31–40.

[52] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[53] "Property Specification Patterns," http://ps-patterns.wikidot.com/.

[54] W. Wei, K. Kim, and G. Fainekos, "Extended LTLvis motion planning interface," in *International Conference on Systems, Man, and Cybernetics*. IEEE, 2016.

[55] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Autonomous Robots*, vol. 38, no. 1, pp. 89–105, 2015.

[56] V. Raman, C. Lignos, C. Finucane, K. C. Lee, M. Marcus, and H. Kress-Gazit, "Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language," University of Pennsylvania Philadelphia United States, Tech. Rep., 2013.

[57] S. Srinivas, R. Kermani, K. Kim, Y. Kobayashi, and G. Fainekos, "A graphical language for ltl motion and mission planning," in *International Conference on Robotics and Biomimetics (ROBIO)*. IEEE, 2013.

[58] U. S. Shah and D. C. Jinwala, "Resolving ambiguities in natural language software requirements: a comprehensive survey," *ACM SIGSOFT Software Engineering Notes*, 2015.

[59] N. Kiyavitskaya, N. Zeni, L. Mich, and D. M. Berry, "Requirements for tools for ambiguity identification and measurement in natural language requirements specifications," *Requirements engineering*, 2008.

[60] J. O. Ringert, B. Rumpe, and A. Wortmann, "A requirements modeling language for the component behavior of cyber physical robotics systems," *arXiv preprint arXiv:1409.0394*, 2014.

[61] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 127140.

[62] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014.

[63] D. Bozhinoski, D. D. Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "FLYAQ: enabling non-expert users to specify and generate missions of autonomous multicopters," in *Automated Software Engineering (ASE)*. IEEE, 2015.

[64] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating CoBlox: A comparative study of robotics programming environments for adult novices," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 366:1–366:12.

[65] G. Biggs and B. Macdonald, "A survey of robot programming systems," in *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, 2003, p. 27.

[66] R. W. Button, J. Kamp, T. B. Curtin, and J. Dryden, *A Survey of Missions for Unmanned Undersea Vehicles.* Santa Monica, CA: RAND Corporation, 2009.

[67] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, 2016.

[68] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II.* Springer, 2013, pp. 214–238.

[69] C. Urmson, C. Baker, J. Dolan, P. Rybski, B. Salesky, W. Whittaker, D. Ferguson, and M. Darms, "Autonomous driving in traffic: Boss and the urban challenge," *AI magazine*, vol. 30, no. 2, pp. 17–17, 2009.

[70] S. Ghosh, D. Sadigh, P. Nuzzo, V. Raman, A. Donzé, A. L. Sangiovanni-Vincentelli, S. S. Sastry, and S. A. Seshia, "Diagnosis and repair for synthesis from signal temporal logic specifications," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, 2016, pp. 31–40.

[71] S. Ghosh, S. Bansal, A. Sangiovanni-Vincentelli, S. A. Seshia, and C. Tomlin, "A new simulation metric to determine safe environments and controllers for systems with unknown dynamics," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 185–196.

[72] W. Li, D. Sadigh, S. S. Sastry, and S. A. Seshia, "Synthesis for human-in-the-loop control systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2014, pp. 470–484.

[73] K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *International Conference on Concurrency Theory.* Springer, 2008, pp. 147–161.

[74] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011).* IEEE, 2011, pp. 43–50.

[75] V. Raman and H. Kress-Gazit, "Explaining impossible high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 94–104, 2013.

[76] ——, "Explaining impossible high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 94–104, 2013.

[77] R. Alur, S. Moarref, and U. Topcu, "Counter-strategy guided refinement of gr (1) temporal logic specifications," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 26–33.

[78] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.

[79] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5382 LNCS, pp. 200–225, 2008.

[80] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[81] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[82] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

[83] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, p. 11, 2018.

[84] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha, "Autonomous vehicles: State of the art, future trends, and challenges," in *Automotive Systems and Software Engineering*. Springer, Cham, 2019, pp. 347–367.

[85] X. Li, Y. Ma, and C. Belta, "A policy search method for temporal logic specified reinforcement learning tasks," *arXiv preprint arXiv:1709.09611*, 2017.

[86] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings fifth ieee international symposium on requirements engineering*. IEEE, 2001, pp. 249–262.

[87] C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Systems of systems engineering: Basic concepts, model-based techniques, and research directions," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 18:1–18:41, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2794381

[88] "Intelligent transport systems - Innovating for the transport of the future," http://ec.europa.eu/transport/themes/its/index_en.htm.

[89] "Current State of EU Legislation - Cooperative Dynamic Formation of Platoons for Safe and Energy-optimized Goods Transportation," http://www.companion-project.eu/wp-content/uploads/COMPANION-D2.2-Current-state-of-the-EU-legislation.pdf.

[90] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, *Software engineering for self-adaptive systems: A second research roadmap.* Springer, 2013.

[91] H. H. Yin Hang, Jan Carlson, "Towards mode switch handling in component-based multi-mode systems." in *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE12)*, Bertinoro, Italy, June 2012., pp. 183–188.

[92] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal 4.0," November 28, 2006.

[93] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e.* IEEE, 1990, pp. 414–425.

[94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and computation*, vol. 111, no. 2, pp. 193–244, 1994.

[95] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres, "Formal verication of autonomous vehicle platooning," February 5, 2016.

[96] J. Rushby, "Just-in-time certification," in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on.* IEEE, 2007, pp. 15–24.

[97] ——, "Runtime certification," in *Runtime Verification.* Springer, 2008, pp. 21–35.

[98] D. Schneider and M. Trapp, "Conditional Safety Certification of Open Adaptive Systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 8, no. 2, pp. 1–20, Jul. 2013.

[99] P. Inverardi, P. Pelliccione, and M. Tivoli, "Towards an assume-guarantee theory for adaptable systems," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 106–115. [Online]. Available: http://dx.doi.org/10.1109/SEAMS.2009.5069079

[100] K. Östberg and M. Bengtsson, "Run time safety analysis for automotive systems in an open and adaptive environment," *SAFECOMP 2013-Workshop ...*, p. NA, Sep. 2013.

[101] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, 2009.

[102] C. Priesterjahn, C. Heinzemann, W. Schfer, and M. Tichy, "Runtime safety analysis for safe reconfiguration," in *IEEE 10th International Conference on Industrial Informatics*, 2012, pp. 1092–1097.

[103] S. Jha and V. Raman, "Automated synthesis of safe autonomous vehicle control under perception uncertainty," in *NASA Formal Methods Symposium*. Springer, 2016, pp. 117–132.

[104] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Reactive synthesis for finite tasks under resource constraints," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 5326–5332.

[105] N. Esfahani and S. Malek, *Uncertainty in Self-Adaptive Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 214–238. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35813-5_9

[106] M. Autili, V. Cortellessa, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli, "Eagle: Engineering software in the ubiquitous globe by leveraging uncertainty," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 488–491. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025199

[107] D. Garlan, "Software engineering in an uncertain world," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 125–128. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882389

[108] T. Everitt, V. Krakovna, L. Orseau, M. Hutter, and S. Legg, "Reinforcement learning with a corrupted reward channel," *arXiv preprint arXiv:1705.08417*, 2017.

[109] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "Torcs, the open racing car simulator," *Software available at http://torcs. sourceforge. net*, 2000.

[110] "Movemo docker image," https://hub.docker.com/r/pmallozzi/rl_monitor/.

[111] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[112] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, July 2015.

[113] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: an automated approach," *Automated Software Engg.*, vol. 14, no. 3, pp. 293–340, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10515-007-0012-6

[114] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, ser. LNCS, no. 1066. Springer–Verlag, October 1995, pp. 232–243.

[115] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[116] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," *Inf. Comput.*, vol. 104, no. 1, pp. 2–34, 1993. [Online]. Available: https://doi.org/10.1006/inco.1993.1024

[117] C. Colombo, G. J. Pace, and G. Schneider, "LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)," in *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, 2009, pp. 33–37.

[118] ——, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2008, pp. 135–149.

[119] D. Dewey, "Reinforcement learning and the reward engineering principle," in *2014 AAAI Spring Symposium Series*, 2014.

[120] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv: ...*, pp. 1–9, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[121] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 1.

[122] E. Wiewiora, "Potential-based shaping and q-value initialization are equivalent," *Journal of Artificial Intelligence Research*, vol. 19, pp. 205–208, 2003.

[123] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete Problems in AI Safety," *arXiv*, pp. 1–29, 2016. [Online]. Available: http://arxiv.org/abs/1606.06565

[124] A. Nowe, K. Van Moffaert, and M. Drugan, "Multi-objective reinforcement learning," in *European Workshop on Reinforcement Learning*, 2013.

[125] C. Liu, X. Xu, and D. Hu, "Multiobjective reinforcement learning: A comprehensive overview," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 3, pp. 385–398, 2015.

[126] E. Letier and A. Van Lamsweerde, "Deriving operational software specifications from system goals," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, p. 119, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=605466.605485

[127] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on software engineering*, vol. 18, no. 6, pp. 483–497, 1992.

[128] "Using Keras and Deep Deterministic Policy Gradient to play TORCS," https://github.com/yanpanlau/DDPG-Keras-Torcs.

[129] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.

[130] P. Mallozzi, R. Pardo, V. Duplessis, P. Pelliccione, and G. Schneider, "Movemo: a structured approach for engineering reward functions," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*. IEEE, 2018, pp. 250–257.

[131] P. Mallozzi, P. Pelliccione, and C. Menghi, "Combining machine-learning with invariants assurance techniques for autonomous systems," in *International Conference on Software Engineering - SE4COG*. IEEE Press, 2018.

[132] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[133] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS 77. USA: IEEE Computer Society, 1977, p. 4657. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[134] L. W. Maxime Chevalier-Boisvert, "Minimalistic gridworld environment for openai gym," https://github.com/maximecb/gym-minigrid, 2018.

[135] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: An automated approach," *Automated Software Engg.*, vol. 14, no. 3, pp. 293–340, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10515-007-0012-6

[136] V. Braberman, N. Kicillof, and A. Olivero, "A scenario-matching approach to the description and model checking of real-time properties," *IEEE Trans. on Software Engineering*, vol. 31, no. 12, pp. 1028–1041, 2005. [Online]. Available: http://publicaciones.dc.uba.ar/Publications/2005/BKO05

[137] D. Harel and A. Kantor, "Multi-modal scenarios revisited: A net-based representation," *Theor. Comput. Sci.*, vol. 429, pp. 118–127, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2011.12.030

[138] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proceedings of ICSE '05*. New York, NY, USA: ACM, 2005, pp. 372–381. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062526

[139] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proceedings of ICSE '08*. New York, NY, USA: ACM, 2008, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368094

[140] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[141] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a java modeling language," in *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998. [Online]. Available: citeseer.ist.psu.edu/\protect\ discretionary{\char\hyphenchar\font}{}{}leavens98jml.html

[142] A. Pnueli, "The temporal logic of programs," in *Proc. 18th IEEE Symposium on Foundation of Computer Science*, 1977, pp. 46–57.

[143] W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider, "A specification language for static and runtime verification of data and control properties," in *FM'15*, ser. lncs. spv, 2015, vol. 9109, pp. 108–125.

[144] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties," in *FMICS'08*, ser. LNCS, vol. 5596. Springer, 2009, pp. 135–149.

[145] G. Reger, H. C. Cruz, and D. E. Rydeheard, "MarQ: Monitoring at Runtime with QEA." in *TACAS*, ser. LNCS, vol. 9035. Springer, 2015, pp. 596–610.

[146] C. Colombo, G. J. Pace, and G. Schneider, "LARVA - A Tool for Runtime Monitoring of Java Programs," in *SEFM'09*, 2009.

[147] T. M. Moldovan and P. Abbeel, "Safe exploration in markov decision processes," *arXiv preprint arXiv:1205.4810*, 2012.

[148] M. Turchetta, F. Berkenkamp, and A. Krause, "Safe exploration in finite markov decision processes with gaussian processes," in *Advances in Neural Information Processing Systems*, 2016, pp. 4312–4320.

[149] P. Thomas, G. Theocharous, and M. Ghavamzadeh, "High confidence policy improvement," in *International Conference on Machine Learning*, 2015, pp. 2380–2388.

[150] Z. C. Lipton, K. Azizzadenesheli, A. Kumar, L. Li, J. Gao, and L. Deng, "Combating Reinforcement Learning's Sisyphean Curse with Intrinsic Fear," *arXiv e-prints*, p. arXiv:1611.01211, Nov. 2016.

[151] W. Saunders, G. Sastry, A. Stuhlmüller, and O. Evans, "Trial without error: Towards safe reinforcement learning via human intervention," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '18. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2018, p. 20672069.

[152] M. Hasanbeig, A. Abate, and D. Kroening, "Logically-constrained reinforcement learning," *arXiv: Learning*, 2018.

[153] M. Al-Shedivat, L. Lee, R. Salakhutdinov, and E. Xing, "On the complexity of exploration in goal-driven navigation," *arXiv preprint arXiv:1811.06889*, 2018.

[154] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," *arXiv preprint arXiv:1604.06057*, 2016.

[155] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[156] L. Medsker and L. C. Jain, *Recurrent neural networks: design and applications*. CRC press, 1999.

[157] J. Randløv and P. Alstrøm, "Learning to drive a bicycle using reinforcement learning and shaping." in *ICML*, vol. 98. Citeseer, 1998, pp. 463–471.

[158] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2018.

[159] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[160] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *CoRR*, vol. abs/1611.05763, 2016. [Online]. Available: http://arxiv.org/abs/1611.05763

[161] IFR, "World Robotic Survey," https://ifr.org/ifr-press-releases/news/world-robotics-survey-service-robots-are-conquering-the-world-, 2016.

[162] Markets and Markets, "Service Robotics Market – Global Forecast to 2022," https://www.marketsandmarkets.com/Market-Reports/service-robotics-market-681.html, 2017.

[163] D. Bozhinoski, D. D. Ruscio], I. Malavolta, P. Pelliccione, and I. Crnkovic, "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective," *Journal of Systems and Software*, vol. 151, pp. 150 – 179, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300317

[164] ISO, "ISO - Robotics," https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-2:v1:en, 2012.

[165] "Roomba Robot Vacuum Cleaners," https://www.irobot.se/roomba.

[166] ABB, "ABB makes robot programming more intuitive with Wizard Easy Programming software," https://shorturl.at/sFU15.

[167] SPARC, "Robotics 2020 Multi-Annual Roadmap," shorturl.at/rIQ07, 2016.

[168] S. Garcia, D. Struber, D. Brugali, T. Berger, and P. Pelliccione, "An empirical assessment of robotics software engineering," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[169] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[170] P. Nuzzo, M. Lora, Y. A. Feldman, and A. L. Sangiovanni-Vincentelli, "CHASE: Contract-based requirement engineering for cyber-physical system design," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 839–844.

[171] P. Nuzzo, J. Finn, A. Iannopollo, and A. L. Sangiovanni-Vincentelli, "Contract-based design of control protocols for safety-critical cyber-physical systems," in *Proc. Design Automation and Test in Europe Conference*, Mar. 2014, pp. 1–4.

[172] C. Menghi, C. Tsigkanos, T. Berger, and P. Pelliccione, "PsALM: Specification of dependable robotic missions," *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*, pp. 99–102, 2019.

[173] J. Krumm, *Ubiquitous Computing Fundamentals*, 1st ed. Chapman & Hall/CRC, 2009.

[174] A. K. Dey, "Understanding and using context," *Personal and ubiquitous computing*, vol. 5, no. 1, pp. 4–7, 2001.

[175] D. D. Bloisi, D. Nardi, F. Riccio, and F. Trapani, *Context in Robotics and Information Fusion*. Cham: Springer International Publishing, 2016, pp. 675–699. [Online]. Available: https://doi.org/10.1007/978-3-319-28971-7_25

[176] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[177] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *CAV*, 2014, pp. 334–342.

[178] Y. A. Feldman and H. Broodney, "A cognitive journey for requirements engineering," in *INCOSE International Symposium*, vol. 26, no. 1. Wiley Online Library, 2016, pp. 430–444.

[179] P. Nuzzo, M. Lora, Y. A. Feldman, and A. L. Sangiovanni-Vincentelli, "CHASE: Contract-based requirement engineering for cyber-physical system design," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 839–844.

[180] R. Darimont and A. Van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6. ACM, 1996, pp. 179–190.

[181] B. DeVries and B. H. Cheng, "Automatic detection of incomplete requirements via symbolic analysis," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 385–395.

[182] ——, "Automatic detection of feature interactions using symbolic analysis and evolutionary computation," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 257–268.

[183] A. Moitra, K. Siu, A. Crapo, H. Chamarthi, M. Durling, M. Li, H. Yu, P. Manolios, and M. Meiners, "Towards development of complete and conflict-free requirements," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, 2018, pp. 286–296.

[184] B. H. Cheng and J. M. Atlee, "Research directions in requirements engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 285–303.

[185] A. Ferrari, F. dellOrletta, G. O. Spagnolo, and S. Gnesi, "Measuring and improving the completeness of natural language requirements," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2014, pp. 23–38.

[186] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *European journal of control*, vol. 18, no. 3, pp. 217–238, 2012.

[187] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[188] Y. Wang, X. Duan, D. Tian, G. Lu, and H. Yu, "Throughput and Delay Limits of 802.11p and its Influence on Highway Capacity," *Procedia - Social and Behavioral Sciences*, vol. 96, no. Cictp, pp. 2096–2104, 2013.

[189] P. Mallozzi, "CoGoMo tool - Web Interface and source code," https://rebrand.ly/cogomoweb, https://rebrand.ly/cogomotool, 2020.

[190] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[191] A. Cimatti, M. Dorigatti, and S. Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 702–705.

[192] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods Symposium*. Springer, 2012, pp. 126–140.

[193] A. Cimatti and S. Tonetta, "Contracts-refinement proof system for component-based embedded systems," *Science of computer programming*, vol. 97, pp. 333–348, 2015.

[194] C. A. Ribeiro dos Santos, A. Saleh, T. Schrijvers, and M. Nicolai, "Condense: Contract-based design synthesis," in *Proceedings of the 22th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Institute of Electrical and Electronics Engineers, 2019.

[195] D. Alrajeh, J. Kramer, A. van Lamsweerde, A. Russo, and S. Uchitel, "Generating obstacle conditions for requirements completeness," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 705–715.

[196] C. Menghi, P. Spoletini, and C. Ghezzi, "Integrating goal model analysis with iterative design," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2017, pp. 112–128.

[197] A. Pneuli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. IEEE, 1990, pp. 746–757.

[198] Y. Lustig and M. Y. Vardi, "Synthesis from component libraries," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 603–618, 2013.

[199] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli, "Constrained synthesis from component libraries," *Science of Computer Programming*, vol. 171, pp. 21–41, 2019. [Online]. Available: https://doi.org/10.1016/j.scico.2018.10.003

[200] P. Mallozzi, P. Nuzzo, P. Pelliccione, and G. Schneider, "Crome: Contract-based robotic mission specication," in *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2020.

[201] C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit, "Provably correct reactive control from natural language," *Autonomous Robots*, vol. 38, no. 1, pp. 89–105, 2015. [Online]. Available: https://doi.org/10.1007/s10514-014-9418-8

[202] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *VaMoS*. ACM, 2019, p. 8.

[203] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[204] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using uml/p statecharts," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 461–466.

[205] M. Klotzbücher and H. Bruyninckx, "Coordinating robotic tasks and systems with rfsm statecharts," *Journal of Software Engineering for Robotics*, 2012.

[206] F.-Y. Wang, K. J. Kyriakopoulos, A. Tsolkas, and G. N. Saridis, "A petri-net coordination model for an intelligent mobile robot," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 4, pp. 777–789, 1991.

[207] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha, "Petri net plans: a formal model for representation and execution of multi-robot plans," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 79–86.

[208] S. Götz, M. Leuthäuser, J. Reimann, J. Schroeter, C. Wende, C. Wilke, and U. Aßmann, "A role-based language for collaborative robot applications," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012.

[209] M. Campusano and J. Fabry, "Live robot programming: The language, its implementation, and robot api independence," *Science of Computer Programming*, vol. 133, pp. 1–19, 2017.

[210] B. Schwartz, L. Nägele, A. Angerer, and B. A. MacDonald, "Towards a graphical language for quadrotor missions," *CoRR*, 2014.

[211] D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Automatic generation of detailed flight plans from high-level mission descriptions," in *International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS. ACM, 2016.

[212] F. Ciccozzi, D. D. Ruscio, I. Malavolta, and P. Pelliccione, "Adopting mde for specifying and executing civilian missions of mobile multi-robot systems," *Journal of IEEE Access*, 2016.

[213] P. Doherty, F. Heintz, and D. Landén, "A distributed task specification language for mixed-initiative delegation," in *Principles and Practice of Multi-Agent Systems*, N. Desai, A. Liu, and M. Winikoff, Eds. Springer Berlin Heidelberg, 2012.

[214] S. Maoz and J. O. Ringert, "Spectra: a specification language for reactive systems," *Software and Systems Modeling*, 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00868-z

[215] K. He, A. M. Wells, L. E. Kavraki, and M. Y. Vardi, "Efficient symbolic reactive synthesis for finite-horizon tasks," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 8993–8999.

[216] S. Moarref and H. Kress-Gazit, "Reactive synthesis for robotic swarms," in *Formal Modeling and Analysis of Timed Systems*, D. N. Jansen and P. Prabhakar, Eds. Cham: Springer International Publishing, 2018, pp. 71–87.

[217] S. Maoz and J. O. Ringert, "On the software engineering challenges of applying reactive synthesis to robotics," in *Proceedings of the 1st International Workshop on Robotics Software Engineering*, ser. RoSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1722. [Online]. Available: https://doi.org/10.1145/3196558.3196561

[218] S. Dragule, S. Garca Gonzalo, T. Berger, and P. Pelliccione, "Languages for specifying missions of robotic applications," *Chapter of the Book Software Engineering for Robotics edited by Ana Cavalcanti, Brijesh Dongol, Rob Hierons, Jon Timmis, and Jim Woodcock*, 2021.

[219] S. Dragule, T. Berger, C. Menghi, and P. Pelliccione, "A survey on the design space of end-user-oriented languages for specifying robotic missions," *Software and Systems Modeling*, 2021. [Online]. Available: https://doi.org/10.1007/s10270-020-00854-x

[220] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel, "Dynamic update of discrete event controllers," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1220–1240, 2020.

[221] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 688699. [Online]. Available: https://doi.org/10.1145/2568225.2568264

[222] P. Mallozzi, P. Nuzzo, and P. Pelliccione, "Incremental refinement of goal models with contracts," in *in submission to Fundamentals of Software Engineering (FSEN) 2021.* IEEE, 2020.

[223] E. A. Lee, *Structure and interpretation of signals and systems.* Lee & Seshia, 2011.

[224] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," 2006.

[225] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification Patterns for Robotic Missions," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[226] P. J. Meyer, S. Sickert, and M. Luttenberger, "Strix: Explicit reactive synthesis strikes back!" in *Computer Aided Verification - 30th*

International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 578–586. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_31